UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

EMULATING NETWORK APPLICATIONS WITH A SOFTWARE-DEFINED NETWORKING ARCHITECTURE

Marcos André Alves Vasco

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

EMULATING NETWORK APPLICATIONS WITH A SOFTWARE-DEFINED NETWORKING ARCHITECTURE

Marcos André Alves Vasco

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada pelo Prof. Doutor Fernando Manuel Valente Ramos
e co-orientado pelo Prof. Doutor Alysson Neves Bessani

2013

Acknowledgments

First and foremost, I want to thank my large family. Starting with my nuclear family, which is my biggest pride and inspiration. I am grateful to my parents for all the love and pride they have, not only in me, but in all their children. It is very gratifying to observe how our achievements are also their achievements. A thank you for all their effort in getting me this far, and for all the investment in my academic formation. To my sisters I give thanks, for all the love, companionship, comprehension, conversations, help, trouble and headaches they have given me over the years. I couldn't have it any other way. Thank you to my brother-in-law, for all the advice he has given me. A special thanks to my grandmother, my dear aunts, my uncles, my grandfather and cousins. Thanks for making our family a Family, with all your love and smiles. Thanks to my godfather, for all the words of strength when I needed, and words of appreciation when I deserved. To my grandmother, aunts, uncles and cousins in Angola, thank you. I love you all, my family.

I wish to thank Cláudia and Zé. To Cláudia, for accompanying me in this stage of my life, and for always being at my side, thank you. To Zé for all his friendship, companionship, and lots of laughs over many years. I also thank him for deciding to enroll in this university, and making my college life even better. Thank you.

I want to thank all my friends from college, namely Tiago Antunes and Carlos 'Guns' Cândido. I have spent a lot of moments with them, working, partying, and chilling. Whatever the situation is, these people should be there, as well as Tiago Ficalho and João Ramos. To the '2ª Fase' class, thank you very much for all the moments we have been through, I loved sharing this college experience with you all. To the people from Lab25, thank you for all your companionship and help.

Last, but not least, I want to thank my advisors. To Professor Fernando Ramos, for accepting me as his student without knowing me, and for all his availability, patience and at ease demeanor in dealing with me, which allowed this year to be more relaxed than it should have been. To Professor Alysson Bessani, for giving me the opportunity to work on an interesting project. To him I owe a word of gratitude for my success in this stage of my life. To my two advisors, I give thanks for allowing me the opportunity to learn from them.

To my Raquelita. You are next.

Resumo

A arquitetura das redes de hoje não é suficiente para responder às necessidades empresariais e académicas correntes. A raiz do problema está na complexidade dos planos de controlo e de gestão (os protocolos e o *software* que coordenam os dispositivos de rede); em particular, devido ao forte acoplamento entre a lógica de decisão e a lógica inerente a um sistema distribuído. Esta complexidade deriva do fato de as redes não possuírem um paradigma de controlo geral, pois não providenciam nenhuma abstração para o seu controlo e gestão.

Software-Defined Networking (SDN) é uma nova arquitetura de rede em que o controlo está desacoplado do reencaminhamento e é diretamente programável. São definidas três abstrações: abstração de encaminhamento, abstração de distribuição de estado, e uma abstração de gestão global. A abstração de encaminhamento permite que *software* de controlo (o controlador) possa comunicar com o plano de dados, através de uma *Application Programming Interface* (API) comum. A abstração de distribuição permite que os programas de controlo de rede não tenham de tratar da disseminação e coleção de estado; o controlador fornece uma *framework* que oferece às aplicações total controlo sobre a rede sem que estas tenham de se preocupar com a forma como as suas decisões se propagam na rede subjacente. Com a abstração de gestão global, as aplicações têm acesso a um modelo lógico da rede, podendo assim geri-la como um único *switch* lógico.

O OpenFlow é uma das primeiras abstrações de encaminhamento em existência, e também a mais comum. É um padrão de comunicação entre o controlador e os dispositivos de rede que cria um ambiente geral de programação, generalizando o plano de dados. O OpenFlow opera nas tabelas de fluxos dos *switches*, de modo a providenciar um protocolo aberto para programar a tabela de fluxos em diferentes dispositivos. Um *switch* OpenFlow comunica com um controlador remoto usando a API OpenFlow, através de uma ligação segura. A tabela de fluxos destes *switches* OpenFlow é povoada com entradas do tipo $\langle$cabeçalho; ações$\rangle$, de acordo com instruções fornecidas pelo controlador. O cabeçalho de um pacote que chega ao *switch* é comparado com os cabeçalhos das entradas na tabela de fluxos e, se houver uma correspondência, as ações correspondentes a essa entrada serão aplicadas ao pacote. As ações suportadas são do género: encaminhar os pacotes para uma certa porta, enviar para o controlador, ou descartar o pacote. Tipicamente, quando um *switch* não consegue encontrar uma correspondência entre um pacote

e as entradas na sua tabela de fluxos, envia o pacote para o controlador.

Os controladores providenciam uma abstração da distribuição de estado. Tal como os sistemas operativos facilitam o desenvolvimento de programas fornecendo acesso controlado a uma abstração de alto nível dos recursos de um computador (memória, processamento, etc.), os controladores são sistemas operativos de rede, que providenciam uma interface de programação uniforme e centralizada para observar e controlar a rede. A interface tem de ser suficientemente generalizada para permitir um amplo espectro de aplicações de gestão. O controlador não efetua a gestão da rede, as aplicações implementadas no controlador são responsáveis por esta gestão. O controlador apenas aplica as ações decididas pelas aplicações. Esta entidade diz-se ser logicamente centralizada pois para as camadas adjacentes esta é abstraída como tal. O controlador não tem de ser centralizado – pode ser implementado, por exemplo, de uma maneira distribuída. O importante aqui é que esta camada esconde a distribuição dos dispositivos de rede, e apresenta uma vista geral da rede para as aplicações.

Numa arquitetura SDN, o plano de gestão é realizado pelas aplicações que correm no controlador, operando sobre uma abstração da rede. Para qualquer requisito de controlo que possa existir, podemos escrever uma aplicação que o resolve. Para o nosso trabalho, iremos construir um *load balancer*, uma aplicação que particiona tráfego por entre vários servidores replicados.

Uma solução para aumentar a capacidade e disponibilidade de um serviço Web é ter múltiplos servidores trabalhando em conjunto como um único recurso. Neste tipo de arquitetura, os pedidos são encaminhados para um dos vários servidores, numa maneira transparente ao utilizador. Num esquema de *load balancing* baseado em *dispatcher*, um dispositivo de rede é colocado no ponto de entrada da rede, que recebe os pedidos e os distribui pelos servidores. Este dispositivo, tipicamente chamado de *load balancer*, age como um *front-end* para o grupo de servidores, e corre um algoritmo de escalonamento para decidir como distribuir a carga por entre os servidores. Soluções correntes para este esquema de *load balancing* são conseguidas através de *hardware* de rede que pode custar alguns milhares de dólares. Para além do mais, estas soluções implementam uma escolha de políticas de escalonamento rígida, com personalização limitada. Outra desvantagem das tecnologias correntes de *load balancing* é que estas permitem apenas particionar a carga por entre os servidores; não podendo portanto escolher o caminho que o tráfego toma até ao servidor. Assim, uma grande vantagem da nossa solução de *load balancing* usando SDN é que podemos escolher o caminho que o tráfego toma para chegar ao servidor. Iremos verificar neste trabalho que a performance do sistema aumenta quando fazemos escalonamento não só de servidores mas também de caminhos.

Desenvolvemos uma aplicação que efetua *load balancing* em servidores conectados numa rede com a arquitetura SDN. A aplicação desenvolvida funciona como um módulo para o controlador POX. Este controlador funciona através de eventos: as aplicações

registam-se como *listeners* de eventos específicos; o controlador faz *raise* a estes eventos, normalmente despoletado por algum acontecimento do plano de dados. Tipicamente, um *switch* que recebe um pacote cujo cabeçalho não corresponde a nenhuma entrada na sua tabela de fluxos, envia este pacote para o controlador. Esta ação despoleta o evento *Packet In* no controlador. A nossa aplicação de *load balancing* está à escuta desses eventos, que normalmente é despoletado pelo primeiro pacote de uma nova conexão. A aplicação escolhe o servidor que irá tratar deste pedido, escolhe o caminho por onde a comunicação se irá dar na rede, e instala regras nas tabelas de fluxos dos *switches* ao longo do caminho. Instala também regras nos mesmos *switches* para o caminho inverso que os pacotes de resposta irão tomar.

Este *Load Balancer* para SDN vem equipado com cinco algoritmos para escolher o servidor e três algoritmos de escolha de caminho. Para escolha de servidor temos os algoritmos: *Round Robin*; *Flow Connections* e *Server Connections* – estes dois escolhem o servidor com menos conexões, diferind no modo como esta informação é recolhida; *Load* – o servidor com menos carga no seu CPU é escolhido; e *Response Time* – o servidor que produz melhores tempos de resposta é selecionado. Os algoritmos de escolha de caminho são: *Shortest Path* – o caminho mais curto (com menor número de *hops*) é escolhido e em caso de empate escolhe-se sempre o de menor identificador; *Equal-Cost Multi-Path* – tal como *Shortest Path*, escolhe-se o menor caminho, mas caso haja mais do que um caminho mais curto, efetua *round robin* entre estes; e *Path Delay* – escolhe-se o caminho que tem o menor atraso.

De modo a poder avaliar os vários algoritmos de balanceamento de carga, escolhemos o emulador Mininet Hi-Fi como ambiente de prototipagem. Este emulador, baseado nos *containers* do Linux, permite a rápida prototipagem de grandes redes usando apenas um computador. Cada nó da rede (*host* ou *switch*) é colocado num *container*, que providencia um ambiente virtual com o seu próprio *namespace* de rede. A rede virtual final obtém-se conectando estes *containers* entre si através de *links* Ethernet virtuais. Os emuladores de redes conseguem correr código real com tráfego interativo, e suportam topologias arbitrárias com um baixo custo. Além disso, o Mininet consegue emular redes que suportem o paradigma SDN. O Mininet Hi-Fi distingue-se dos demais emuladores pois providencia recursos para se obter fidelidade de desempenho. Esta é conseguida usando mecanismos de isolamento e provisão de recursos, e monitorizando a experiência para verificar se esta corre de uma maneira realista.

Criou-se uma topologia baseada na topologia de rede *fat-tree*, com 4 servidores, 5 *switches* e 2 clientes. A experiência consistiu em ter os clientes a efetuarem um elevado número de pedidos aos servidores. Os servidores, ao receberem estes pedidos, efetuam processamento que impõe carga variada no seu CPU, de modo a simular pedidos diferentes. Os clientes registam o tempo de resposta de cada pedido. Todos estes valores foram agregados e apresentados em diagramas de caixa (*boxplots*). Esta experiência foi

repetida com diferentes distribuições para a variação da carga da CPU e para diferentes configurações dos algoritmos. Para todas as situações, é visível uma melhoria significativa quando se efetua escalonamento de carga entre os vários caminhos disponíveis. Isto verifica-se independemente do algoritmo de escolha de servidor. Concluímos, então, que uma solução em SDN consegue não só ser melhor em termos de desempenho, mas também sair mais em conta em relação a custos de equipamento.

Palavras-chave: Software-Defined Networking, Mininet, Multi-path, OpenFlow, Load Balancer

Abstract

Current network architectures are ill-suited to meet today's enterprise and academic requirements. The problem lies in the complexity needed to control and manage the network. This complexity stems from the strong coupling between the control and data planes. A novel network paradigm, Software-Defined Networking (SDN), was proposed to alleviate this problem. The main idea is to decouple the control and data planes, allowing the network to be programmatically controlled. A key entity in SDN architectures is the controller. This logically centralized entity acts as a network operating system, providing applications with a uniform and centralized programming interface to the underlying network.

In this Thesis we will develop an application that performs server load balancing in a Software-Defined Network. Today's load balancers are high-priced devices that have limited customizability. Furthermore, they can only balance load between the servers, and are forced to use the path the network provides. In an SDN architecture, performing server load balancing does not require these expensive and inflexible devices. A low-cost load balancing solution can be designed, further enhanced with the ability to schedule load not only between servers, but also between paths. In addition, the centralized network control permitted in SDN allows the collection of updated link information to be used by the same logically centralized entity that makes forwarding decisions, which permits dynamic scheduling algorithms to decide on a best path for network traffic. We will show that such a solution outperforms ones which are oblivious to the path the traffic takes.

Keywords: Software-Defined Networking, Mininet, Multi-path, OpenFlow, Load Balancer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer networks are an integral part of modern society. They are the backbone of all web services used today, and have become crucial in the infrastructure of our businesses, homes and schools. An ever increasing number of devices are connected to the internet, and thus connected to a computer network. This work aims to reflect on a new network paradigm, called Software-Defined Networking (SDN), and its benefits in the control and management of a computer network.

1.1 Traditional Computer Networks

In current computer networks, functionality can be split into three main planes: data, control, and management. The data plane handles packet forwarding. Network devices route packets to their destination, sending them to the next-hop device along the path by forwarding the packets to one of its ports. Devices use information on their forwarding tables and on the packet's headers to decide which port to send the packets to. The control plane consists of the distributed routing algorithms that fill the forwarding tables. Routers in interconnected networks exchange information about destination addresses using dynamic distributed routing protocols. These devices must therefore understand and process a myriad of protocols. The management plane monitors the network and configures the data plane mechanisms and control plane protocols. A view of the three planes and how they are implemented in today's networks can be seen in the left part of Figure 1.1. In it, we can see how the control and data planes are coupled in the switches. This means that all the functionality pertaining these two planes is achieved by all network devices working collectively.

1.1.1 Challenges

Computer networks have become part of the critical infrastructure of businesses, schools and homes, with an undeniable success. However, the original computer network architecture is ill-suited to meet today's requirements. When the mechanisms used in current

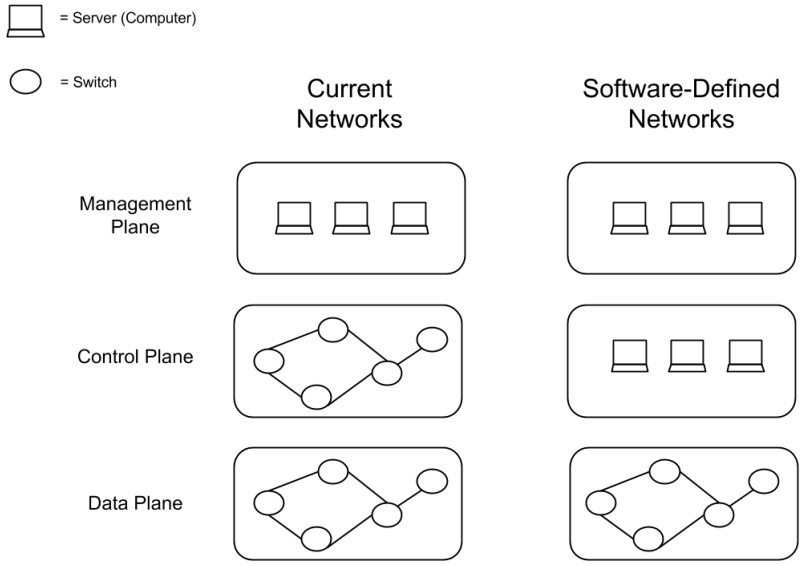
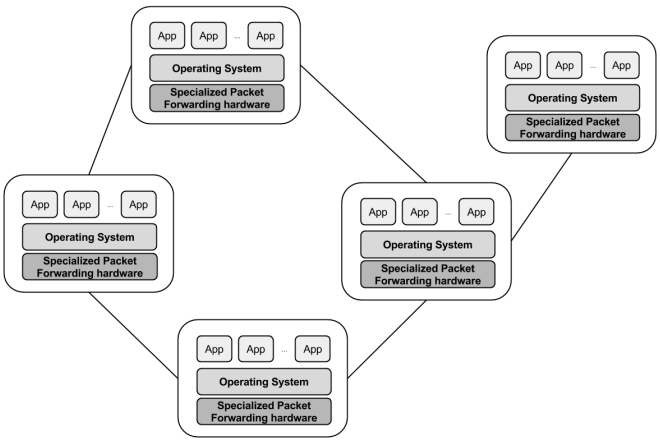


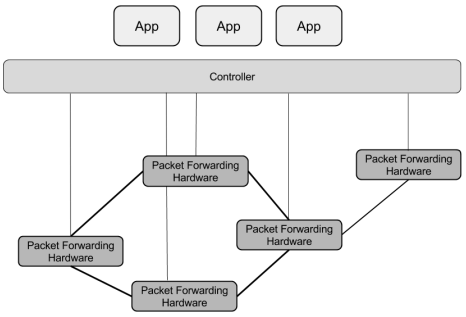
Figure 1.1: The management, control and data planes on current network architectures and SDN architectures.

networks were designed, they were simple. The original Internet Protocol (IP) control plane was designed to have a single distributed algorithm to maintain the forwarding tables in the data plane [15]. Nowadays, however, things are more complex. Protocols have evolved to deliver higher performance, reliability, broader connectivity and rigorous security. These protocols tend to be defined in isolation, with each solving a specific problem without the benefit of any underlying abstractions. The result of this is the primary limitation of today's networks: complexity [30]. The TCP/IP and Open Systems Interconnection (OSI) layer models are good abstractions, but they only deal with the data plane. One of the main problems with current networks is a lack of powerful control plane abstractions [37].

In addition, the control requirements for today's networks — Access Control Lists (ACLs), Virtual Local Area Networks (VLANs), middleboxes, etc. — add complexity to the operation and management of a network. These requirements lead to incremental changes in control plane protocols and complex management plane software, which results in ever growing complexity [15]. State and parameters of network components are distributed across the network, making it difficult to guarantee state consistency among all network devices. This leads to management solutions that are both expensive and error-prone. For example, to implement a network-wide policy, possibly hundreds of devices and mechanisms must be configured. The complexity of today's networks and the inherent difficulties in managing them makes it very difficult to apply a consistent set of access, security or Quality of Service (QoS) rules. This can result in network vulnerability to security breaches and enormous struggles for network managers [7, 30].



(a) Closed Network Architecture



(b) Opened Network Architecture

Figure 1.2: Logical view of the architecture of current networks and SDNs.

Current networks are relatively static as a result of their complexity, making it hard for them to innovate and to cope with several challenges. For example, due to this static nature, they cannot dynamically adapt to changing traffic, application and user demands [30]. Another consequence of the complex network architecture is the difficulty to scale. Every time a new network device is added to the network, it must be configured, and possibly several other devices must be reconfigured as well. To make matters worse, network devices from different vendors usually have different interfaces, which makes it more difficult to configure and manage all the devices in the network.

1.2 SDN: A New Network Paradigm

Due to the fundamental difficulties to operate and manage network control components using a distributed approach, in recent years some work has been done in refactoring the network control plane in a centralized approach [5, 7, 15]. A new networking paradigm, SDN, originated based on such ideas.

In Software-Defined Networks, the control plane is separated from the data plane. It is moved out of the individual network devices, to be implemented in software in separate servers. This allows the network to be programmatically controlled [30]. Network devices become simple, vendor-independent, packet forwarding devices that no longer need to understand and process the wide range of protocol standards used in current networks. They simply receive packet forwarding and control instructions from a specialized entity (a network controller or network operating system). Network applications, such as routing, load balancing, etc., run on this logically centralized controller, as seen in Figure 1.2(b). This is in clear contrast with current networks, where the control logic is embedded in all the network devices (Figure 1.2(a)). The controller maintains a global view of the network, supplying programmers with a simplified and programmatically configurable network abstraction. This makes it possible to easily implement and manage a wide range of network services, such as routing, access control, QoS, etc. [30]

We refer to the controller as a logically centralized entity because it is abstracted to both the applications and the switches as such. A physically centralized solution is likely to arise scalability concerns. However, the controller can be transparently implemented in a distributed fashion. As such, scalability issues can be reasoned about and tackled similar to any distributed system [40].

1.3 Motivation

In this work, we will develop and evaluate an SDN control application: a load balancer. Such a load balancer is accomplished using an inexpensive software application that runs on an SDN controller. Therefore, this low-cost solution is in stark contrast with current load balancing solutions, which are materialized by expensive network devices. We aim to show how simple it is to write a control application that performs load balancing custom-tailored to a specific scenario, as opposed to purchasing an expensive and rigid load balancing hardware. Furthermore, an SDN load balancing solution has the added benefit of considering network status when scheduling load. Different paths can be chosen to accommodate different network congestion scenarios, providing added flexibility and increased performance.

1.4 Contributions

Our work's main contribution is the development and evaluation of a load balancing solution for a Software-Defined Network. We have:

- Implemented a load balancer in an SDN;

- Evaluated and compared typical load balancing algorithms (round robin, number of connections, server load);
- Evaluated a novel algorithm that takes advantage of the ability to take network load into account when scheduling traffic. Current load balancing solutions can only choose the target server for a request, and are forced to use the path the network provides. In our work, thanks to the network control permitted by an SDN architecture, our application can partition load between not only the servers, but also the paths traffic takes to get to the chosen server. This way we can effectively take network load into account when scheduling traffic, as opposed to taking only server load into account. This can be particularly beneficial in datacenter topologies, which are made to have multiple paths to a resource in the network;
- Tested and evaluated our load balancing control application using the Mininet Hi-Fi [18] network emulator. This emulator is used in virtually all SDN-related work. With Mininet Hi-Fi, we can emulate large networks in a single machine that support the SDN paradigm. The virtual software switches connect to a remote SDN controller and can communicate using the OpenFlow [25] protocol. In addition to its support of Software-Defined Networks, Mininet Hi-Fi offers more realism and fidelity than simulators, and is cheaper, simpler and more flexible than a test bed.

1.5 Work Plan

Initially, our work plan consisted in:

- Task 1 (October – December 2012) — Literature survey and writing a preliminary report;
- Task 2 (December 2012) — Learning to use Mininet Hi-Fi;
- Task 3 (January 2013) — Emulate a load balancer using Mininet Hi-Fi on a Software-Defined Network
- Task 4 (February – April 2013) — Emulate JITeR [11] using Mininet Hi-Fi on a Software-Defined Network;
- Task 5 (April – June 2013) — Writing of the project's report

The plan was slightly changed as we progressed in our work. We abandoned the idea of developing an application that mimics the behavior of JITeR due to three factors: 1) Mininet Hi-Fi was lauched in mid December, until then we worked with the previous version of Mininet that raised fidelity concerns. This caused a delay in the writing of a

preliminary report, and in all subsequent work; 2) we saw enough significance in extending the study of the load balancer, in particular due to the possibility of partitioning traffic among multiple network paths, something we have not anticipated; 3) we saw the need to run a significant number of lengthy experiments of our load balancer application.

1.6 Document Structure

This document is organized as follows:

- Chapter 2 — Here we provide a discussion of the related work. We break down the structure of an SDN architecture and explain its inner workings. We describe the abstractions this novel network architecture provides and detail the implementations of the ones used in our work. Since we aim to create a load balancer, we give an explanation of what load balancing is, and how it is usually accomplished in today's networks. We also provide a detailed description of the emulator we use in our work, Mininet Hi-Fi, and how it works.

- Chapter 3 — This chapter describes the design of the load balancer and the algorithms evaluated and proposed.

- Chapter 4 — Here we evaluate the different load balancing algorithms. We present the network topology used, and provide a detailed description of the experiments. We provide a discussion on how we set up Mininet Hi-Fi to accurately run our experiment, and show the results of several performance measures

- Chapter 5 — In the last chapter we present the conclusions we take from this work. We also present a discussion on things that can be improved as future work.

Chapter 2

Related Work

In this work we investigate how Software-Defined Networking (SDN) can improve network control and management. This chapter aims to detail the concepts and tools used in our work.

We will try to elucidate what SDN is, how it is accomplished, and how network managers may benefit from it.

To convey how relatively simple it is to solve network control problems in an SDN architecture, an application that performs load balancing was developed. We will describe what load balancing is, how it can be useful, and how it is implemented in today's networks.

As we wish to study how our load balancing solution performs, we deployed it in an emulated network environment provided by the Mininet Hi-Fi tool. We will unravel what this network emulator can do, how it works, and why it is useful in our work.

2.1 Software-Defined Networking

In this new network architecture, control is decoupled from the network devices, and is directly programmable. The network devices become simple packet forwarding devices, which receive control instructions from a *logically* centralized entity known as the controller. By logically centralized we mean that control logic is to be designed and operated as if it was a centralized application, rather than a distributed state [24]. However, the controller itself may be a distributed system, as is in fact the case with production SDNs, such as Google's private Wide Area Network (WAN) [21].

Current networks have no powerful control plane abstractions. SDN aims to solve this problem. The control plane is redefined as three abstractions: a forwarding abstraction, a state distribution abstraction and a global management abstraction. The forwarding abstraction allows a software controller to communicate directly with the data plane, using a common Application Programming Interface (API) to program the network hardware. In SDNs, the materialization of this abstraction is most commonly done using Open-

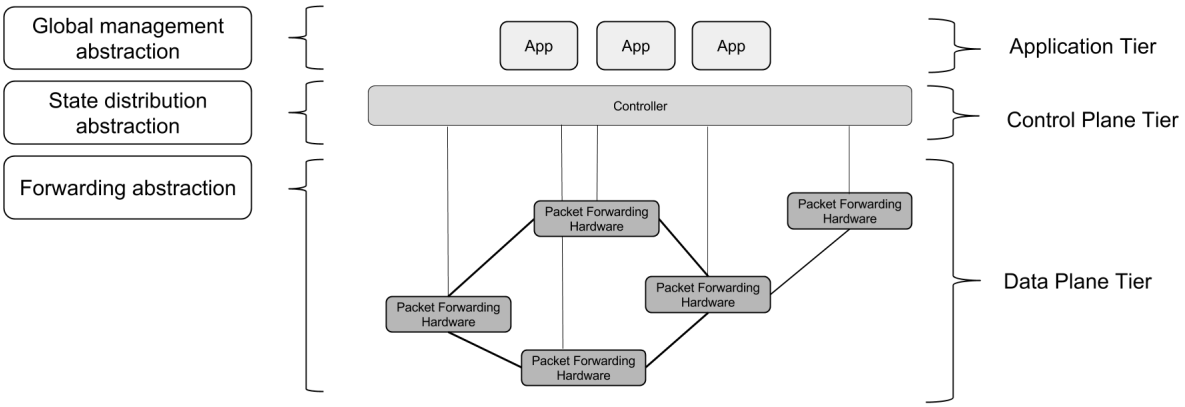


Figure 2.1: SDN abstractions and architecture.

Flow [25]. The state distribution abstraction shields control programs from the vagaries of distributed state. Thus, management applications no longer have to worry about dissemination and collection of state. The logically centralized controller accomplishes the state distribution abstraction. With the global management abstraction the network has a logical appearance and can be managed as a single logical switch, rather than having to program each individual network device one at a time [37].

The network becomes divided in three tiers, as seen in Figure 2.1. The switches — now "dumb" packet forwarding devices — are located in the data plane tier; the controller and the network applications are in the control plane and application tiers, respectively.

2.1.1 OpenFlow

OpenFlow [25] is the most common forwarding abstraction in SDNs. It is the first standard communications interface defined for the exchanging of information between the controller and the packet forwarding devices. While it is not mandatory to use OpenFlow, it is nowadays the most common standard used for the communication between SDN controllers and packet forwarding devices.

OpenFlow started out as a way for researchers to run experimental protocols in networks used every day. As explained before, networks today are static. A lot of the algorithms that are used, as well as functions, are fixed in hardware, in the network device's chips. This results in a high barrier of entry for new ideas, due to the enormous installed base of equipment and protocols. Commercial solutions, meaning proprietary equipment, are closed and inflexible. Research solutions on the other hand, either have insufficient performance or are too expensive. OpenFlow, however, attempts to have switches support a broad range of applications, with high performance and low-cost implementations, all while being consistent with vendor's need for closed platforms.

OpenFlow operates on the switches' flow tables. While each vendor's flow table may

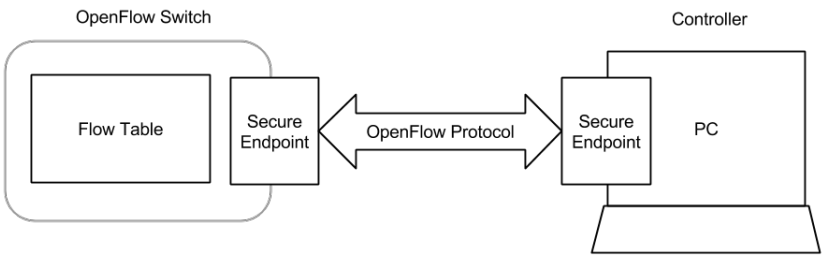


Figure 2.2: OpenFlow Switch's Components. The flow table is managed by a remote controller using the OpenFlow protocol, via a Secure Channel [25].

be different, OpenFlow exploits a common set of functions that run in many network devices. The goal is to provide an open protocol to program the flow table in different network devices. This way, network traffic can be partitioned into production traffic and research traffic. Flows can be controlled, the paths that packets follow can be chosen, as well as the processing they receive. OpenFlow can be compared to the instruction set of a Central Processor Unit (CPU), since it specifies basic primitives that can be used by external software (in SDN, the controller) to program the forwarding plane of the network devices.

We present the three building blocks of an OpenFlow switch in Figure 2.2: a flow table, with an action associated with each flow entry; a secure channel connecting the switch to a remote controller; and the OpenFlow Protocol, which provides an API for the controller to communicate with the switch. The flow table is populated with flow entries of the form $\langle\ header; action\ \rangle$, as decided by the remote controller. Packet headers are compared with the header field of flow entries on the switch. If there is a match, the action associated with the matched entry is performed on the packet. The switch does not have to know what it means in term of distributed state, it only knows what it is supposed to do.

There are three basic actions that all OpenFlow switches must support: forward the flow's packets out to a certain port (or ports) so they can be routed through the network; encapsulate and forward a packet to the controller, typically used when there are no matches in the flow table (for the first packet of a flow), so the controller can decide on an action; and drop the flow's packets. The OpenFlow protocol also allows for the modification of various packet header fields, such as source and destination addresses and ports. Counters are also maintained for each flow entry, recording the number of matching packets and bytes transferred.

MAC **src**	MAC **dst**	IP **src**	IP **dst**	TCP **dport**	...	**Action**	**Count**
*	*	10.1.1.2	192.168.*	35671	*	port 1	1098
*	10:20:.	*	*	*	*	port 2	250
*	*	*	5.6.7.8	*	*	port 3	300
*	*	*	*	25	*	drop	120
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Table 2.1: An example illustration of an OpenFlow-Enabled Switch's flow table [30].

A flow is a broad definition, limited only by the capabilities of the implementation of the flow table. This means that a flow can be a Transmission Control Protocol (TCP) connection, or packets from a particular Media Access Control (MAC) address, or packets with the same VLAN tag, and so forth. This can be observed in Table 2.1, which presents an illustration of a flow table with a subset of its various fields. Rules are ordered in priority. Wild carded fields (represented by *) enable a wide range of policies to be implemented in the network. As an example, the third rule in Table 2.1 will match on all packets with an IP destination address of $5.6.7.8$, regardless of the remaining fields of the packet. The ability to allow a network to be programmed on a per-flow basis provides extremely granular control, which enables the network to respond to real-time changes at the application, user and session levels [30]. On OpenFlow-enabled switches there can be an additional action one can exert on a flow: to forward the flow's packets through the switch's normal processing pipeline. This allows isolation between experimental traffic and production traffic [25].

While the initial goal of OpenFlow was to foster innovation, allowing one to easily experiment a network protocol by dividing production and experimental traffic, the greater value of the OpenFlow API is that it creates a general programming environment for the data plane of a network. By generalizing the forwarding path it allows the decoupling of the distribution model from the control logic on the network elements.

2.1.2 SDN Controllers

Controllers offer a uniform and centralized programmatic interface to the entire network. Much like operating systems provide controlled access to high-level abstractions for the resources of a computer system, thus facilitating program development, the controller software is a "network operating system", providing the ability to observe and control a network. The interface offered must be general enough to support a broad spectrum of network management applications [25].

The controller does not manage the network itself; applications implemented on top

of it perform the actual management [16]. The controllers form the control plane of the SDN network and the applications form the management plane. For example, the controller merely adds and removes flow-entries from the switches' flow tables on behalf of the network management applications [25].

Some concerns about availability and scalability may arise when devising a network architecture based on a centralized controller. However, enough resilience can be achieved by applying standard replication techniques [16]. In fact, the term *logically centralized* is an oversimplification. What is important to note is that the distribution model is our choice to make, and not the network's choice [37]. Thus, the controller can be a distributed system built and configured based on the specific requirements of scalability, resiliency, availability, etc. [22] All that is needed to maintain is a unified network view. As with any distributed system, the choice in consistency model offers a tradeoff between performance and overhead, which influences SDN scalability [40]. Furthermore, the OpenFlow protocol allows for a switch to be controlled by more than one controller, for increased performance and resilience [25].

Controllers present programs with a centralized programming model, allowing applications to be written as if the entire network were present on a single machine. This is made possible by logically centralizing the network state. Controllers also allow programs to be written in terms of high-level abstractions, such as users and host names, instead of low-level configuration parameters, like IP and MAC addresses. Management rules can be enforced independent of the network topology; provided the controller maintains mappings between these abstractions and the low-level configurations [7, 16].

There is already a significant number of controllers available to choose from: NOX [16], POX [32], Floodlight [12], Ryu [33], etc. We have decided to use POX, a Python version of the first SDN controller: NOX.

NOX

NOX was the first SDN controller made available. It tries to provide a modular and flexible framework for users to write control components that achieve control plane goals, using OpenFlow switches. A NOX-based network consists of a set of switches and one or more network-attached servers, running the NOX controller software and the management applications over it. The NOX software consists of several different controller processes (one per server) and a single network view, kept in a database running on one of the servers. The network view contains the results of NOX's network observations, and applications use this state to make management decisions [16].

NOX provides observation granularity at the switch-level topology, showing the locations of users, hosts, middleboxes and other network elements [16]. As for control granularity, once control is enforced on some packet, subsequent packets with the same header can be treated the same way, meaning that control granularity is done at the flow level.

These granularity choices allow the system to scale to relatively large networks while still providing flexible control. The only component that is global (must be consistent across controller processes) is the network view.

When an incoming packet's header does not match any flow entry at a switch, it is forwarded to the controller process. NOX uses these packets to construct the network view and applications running on NOX use them to determine the control actions to exert on the network. Typically a packet whose header does not match any flow entries is the first packet of a new flow. If the application decides to install an entry for this new flow, subsequent packets will match on the flow entry installed; the switch updates the appropriate flow counters and applies the corresponding actions.

Since NOX is responsible for establishing every flow in the network, it may become a bottleneck if it does not have enough capacity to handle all the requests. However, a single controller process can handle 100000 flow initiations per second [16], which is considered enough for a good range of networks [3].

NOX's interface revolves around events, a namespace and the network view. Applications use a set of handlers registered to execute when a particular event happens, typically triggered by the dataplane. Applications register for events, and NOX invokes a handler when an event occurs, processing each event individually. The application handler's return value indicates to NOX whether to stop the execution of an event, or to trigger the next registered handler. Events can be generated either in direct response to received OpenFlow messages, or by the applications as they process other events.

POX

POX [32] is the Python version of NOX. Its focus is on research and academia. NOX's core infrastructure was implemented in C++; however, it allowed applications to be written in either Python or C++. As development of NOX progressed, developers saw the need to build separate Python and C++ versions. So POX was forked from NOX, but the basic idea and framework remains the same [26].

2.1.3 Management Applications

On top of the control plane resides the management plane. On an SDN architecture, the management plane is accomplished by the software applications operating on an abstraction of the network state. For all the control requirements that may exist in current networks, we can write management applications to satisfy them.

The control problem which we will develop an application for is network load balancing. We will construct a load balancer for a Software-Defined Network.

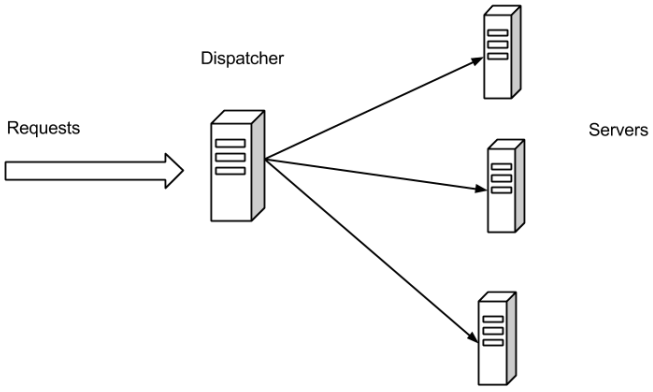


Figure 2.3: High-level view of a dispatcher-based web server cluster architecture.

2.2 Load Balancing

Today's internet has a high, ever-growing volume of traffic. Some online services, such as web sites and social networks, have to be able to cope with millions of requests per day. The need to improve web server capacity is often faced by administrators. A popular solution to increase capacity and improve reliability of web services is to have multiple servers working collectively as a single web resource. Online services are often replicated on multiple servers, providing greater capacity and increased reliability. This collection of servers can be known as a distributed web server, a server cluster, or a web farm [34]. In this distributed architecture, incoming requests are routed among the various server nodes in a user-transparent way.

To effectively utilize the resources of a cluster of servers, a load balancing scheme must be adopted. The chosen scheme consists of the entity doing the load balancing and the algorithm used to decide which server to send each request to. A dispatcher-based load balancing approach is characterized by having a separate entity — the dispatcher — deployed at the network's ingress point, receiving the incoming requests and distributing them among the servers [6], as seen in Figure 2.3. A specialized network routing device, called a load balancer, typically materializes the dispatcher. The load balancer acts as a front-end for the server cluster, and runs a scheduling algorithm to make decisions regarding the distribution of load among the servers in the cluster. The algorithm plays an important part in the effectiveness of this distributed solution, since the choice of the right server to assign each request can increase cluster throughput and decrease the mean response time.

Load balancing is a technique used to evenly distribute workload amongst a collection of resources, in order to get optimal resource utilization, minimum response time,

maximum throughput and avoid overload. In a typical dispatcher-based approach the server cluster is presented to the clients as a single virtual IP address, or IP alias. This is typically the address of a dispatcher that acts as a centralized scheduler and controls all routing decisions of all client requests. An example communication pattern in this approach would be: 1) a client sends a request to the IP address alias for some online service; 2) the request reaches the dispatcher; 3) the dispatcher selects the server that will service the request using some scheduling policy, and forwards the request to the chosen server in the cluster; 4) the chosen server handles the request and replies to the client. This distributed architecture allows a service to be easily scalable, since it is possible to add additional server machines to augment the performance of the service. Another benefit is service availability. An appropriate scheduling algorithm can provide the service with fault resilience, allowing machines to leave the cluster due to failure or maintenance, with a graceful degradation of performance [34].

2.2.1 Current Load Balancing Solutions

Today's dispatcher-based load balancing solutions are accomplished using a vendor-specific network routing hardware (the dispatcher). These components have become essential in modern networks, and have evolved to provide other functionalities besides load balancing, like caching and some network security features.

However, there are some disadvantages with this approach. Dedicated load balancing hardware can be expensive (typically several thousand dollars). This can be an even greater burden if a service/network needs more than one, as is common. Furthermore, dispatchers run vendor software that implements a rigid choice of policies, and have limited customizability [39]. It can be impossible to tailor scheduling policies to a particular need, and specialized administration may be required. On unstructured networks, such as a WAN, it can be hard to know what the best locations for the dispatchers are in order to effectively partition network load.

An important limitation of current dispatcher-based approaches is that the load can only be partitioned between the servers; dispatchers cannot choose the path the requests follow. They are forced to use the path imposed by the network. This can be a relevant disadvantage when there are multiple paths to a resource. This is a disadvantage we can tackle using SDN. Since we have full control over the network, we have the capability of not only choosing the destination of traffic but also the path it takes.

2.3 Emulating Software-Defined Networks

In order to evaluate our network control application, we opted to use a network emulator. The tool we use to make such emulations is Mininet Hi-Fi [18], an extension of Mininet [23].

2.3.1 Mininet

Mininet is a container-based emulator, employing lightweight OS-level containers that share a single kernel. It allows for the rapid prototyping of large networks on the constrained resources of a single machine [18]. Mininet differs from other emulators employing lightweight virtualization in its support of Software-Defined Networks. A consequence of an SDN architecture is that the functionality of the network can be defined after it has been deployed. New features can be added in software, without modifying the switches. Systems prototyped on Mininet support this paradigm. Mininet has been used as a prototyping environment in just about all SDN-related work, being now the *de facto* emulator used in such projects.

Current available prototyping environments have their pros and cons. Special-purpose test beds for networking can be very realistic, but they are costly to build and maintain, have practical resource limitations and may lack the flexibility to support experiments with custom topologies or behavior. The high cost alone makes it beyond the reach for most researchers [23]. Simulators, such as NS-2 [28], are appealing because they can run on a single machine. However, they lack realism. Their models for hardware, protocols and traffic generation may raise fidelity concerns. Moreover, the code created on the simulator is not the same code that would be deployed on a real network, and they are not interactive. One could envision a solution with a network of virtual machines, having a virtual machine per network element and host. However, virtual machines are too heavyweight, limiting the scale of such a network to just a handful of switches and hosts.

Emulators, nonetheless, can obtain the best of both worlds. Like test beds, emulators run real code with interactive network traffic; and like simulators, they support arbitrary topologies with low cost. Furthermore, emulator code can be "shrink-wrapped" and ported to a virtual machine, allowing one to effortlessly share their experiments [18]. Since Mininet preserves switch, application and script semantics between emulation and hardware, an idea that works on Mininet can be deployed on a real production network for validation or general use.

However, emulators may not provide adequate performance fidelity for experiments. As CPU resources are multiplexed in time by the scheduler, there is no guarantee that a host ready to send a packet will be scheduled immediately, or that virtual software switches will transmit data at the same rate as their physical counterparts. This is due to background loads that affect the performance of virtual nodes, leading to unrealistic results. Infidelity may arise when multiple processes execute serially on the available cores, rather than in parallel as in physical hosts and switches. Unlike simulators, Mininet runs in real time and does not pause a host's CPU clock. Therefore, events such as transmitting a packet or finishing a computation are susceptible to delays, due to serialization and background system load [18].

Mininet's original architecture uses lightweight OS-level virtualization to emulate network elements. It makes use of Linux containers, a virtual system mechanism built on top of the kernel. It provides a virtual environment that has its own network namespace. A virtual network is created by placing hosts in a separate container (with a separate network namespace) and connecting them with virtual Ethernet pairs.

By using lightweight virtualization, Mininet can scale to hundreds of nodes and still maintain interactive performance. It was shown in [23] that networks with hundreds of switches can be started in tens of seconds. Topologies with a significant number of switches and hosts cannot fit in memory using system virtualization. Such large networks can be emulated on Mininet because it virtualizes less and shares more: the file system, user and process spaces, kernel devices and libraries are shared between containers and managed by the Operating System (OS).

Mininet provides a Python API that allows the creation of custom topologies. Switches, hosts, links and controllers can be defined and custom-tailored in a few code lines of Python.

The main problem of the original Mininet is that it does not provide any assurance of performance fidelity, because it does not isolate the resources used by virtual hosts and switches.

2.3.2 Mininet Hi-Fi

Mininet Hi-Fi attempts to accurately emulate and reproduce experiments limited by network resource constraints on a network of hosts, switches and links. This is done by carefully allocating and limiting host CPU and link bandwidth, and then monitoring the experiment to ensure the emulator is operating within the limits imposed, producing realistic results [18]. Thus, Mininet Hi-Fi extends the original Mininet architecture, adding mechanisms for performance isolation, resource provisioning and monitoring for performance fidelity.

Mininet Hi-Fi implements CPU and network bandwidth limits using OS-level features in Linux. As in Mininet, it uses control groups, or *cgroups*, to group processes together (belonging to a container/virtual host). It then enforces limits and isolation on the resource usage of each *cgroup*. CPU bandwidth is limited by enforcing a maximum time quota for a *cgroup* within a given period of time. Traffic control is exerted using the *tc* command, which allows the configuration of network link properties, such as bandwidth, delay, and packet loss.

Resource provisioning is accomplished by splitting the CPU among containers, with some margin to handle packet forwarding. Since the exact CPU usage for packet forwarding varies (with path length, lookup complexity and link load), it is hard to know in advance the correct configuration for an experiment. It is up to the experimenter to allocate link speeds, topologies and CPU slices based on an estimated demand.

Mininet Hi-Fi relies on the monitoring of the performance fidelity to help verify if an experiment is operating realistically. Measuring the inter-dequeue times of packets monitors link and switch fidelity. As links run at a fixed-rate, packets should leave at predictable times whenever the queue is non-empty. To monitor host fidelity, the CPU idle time is observed. CPU bandwidth limiting ensures that no virtual host receives excessive CPU time, but not whether each virtual host is receiving sufficient time to execute its workload. The presence of idle time implies that a virtual host is not starved for CPU resources, and the absence of idle time is conservatively assumed to indicate that fidelity has been lost and the experiment should be reconfigured [18].

Mininet Hi-Fi is ideal for experiments that have network constraints, that is, are limited by network properties such as link bandwidth, rather than other system properties such as memory latency, and have aggregate resource requirements that would fit within a single modern machine. It is therefore a good fit for experiments that have modest resource requirements [18].

2.4 Final Remarks

This chapter provided a detailed description of the key concept of our work, SDN. We have seen how the separation of the control and data planes is achieved, and the inner workings of the controller software we will use to build of our SDN load balancer application. We have also explained what load balancing is, and how it is accomplished in current networks. As our choice in prototyping environment lied in Mininet Hi-Fi, we have justified that choice, and detailed how this emulator achieves its goals.

Given the background information provided in this chapter, we are now able to detail how a load balancer application can be built in a Software-Defined Network. In the next chapter we will describe how we have built our load balancing solution, as well as its scheduling algorithms.

Chapter 3

Load Balancing in SDN

Built as an application on top of a controller, load balancing behavior can be achieved without the need for expensive commercial hardware. Due to the extent of the network control allowed in a Software-Defined Networking architecture, it is possible to augment a load balancing scheduling policy to consider not only server load, but also network link load as well, as seen in previous work [19].

3.1 The Load Balancer Application

We have created a load balancer as an application built on top of the POX controller. As an OpenFlow controller, POX is connected to all the OpenFlow-enabled switches in the network. It treats applications as modules that can be loaded when it starts. These modules listen for and handle events triggered by POX in result of the switch's behavior. The application is built by listening to specific events, and enforcing a set of network actions to be executed by the switches.

Message	Description
Flow Modification	Edits the flows on a flow table. We use it to add an entry to the flow table, but it can be used to edit or delete a flow entry.
Statistics Request	Requests statistics from the switches. We use it to request flow table statistics, but it can be used for port, queue and switch statistics, amongst others.
Set Configuration	Edits OpenFlow configuration options on a switch.

Table 3.1: The OpenFlow messages used in our application.

Event	Description
Connection Up	When a connection to an OpenFlow switch has been established.
Packet In	When a packet with no matching flow arrives at a connected OpenFlow switch.
Flow Statistics Received	When a message containing a switch's flow table statistics arrives. This message is received because the application has previously requested the switch to do so (see *Statistics Request* in Table 3.1).

Table 3.2: The OpenFlow events our application handles.

OpenFlow's API is extensive [29]; however, we are only interested in a handful of operations. As seen before, the controller and the switches communicate through a secure channel. The controller exerts the application's actions on a switch by sending it special-purpose OpenFlow messages, and presents the application with switch data triggering OpenFlow events. OpenFlow events contain attribute objects that hold the respective switch's information and the OpenFlow message that triggered the event. For our purposes, we need to consider three OpenFlow messages, and need to listen to three events only, as reported in Tables 3.1 and 3.2, respectively.

The default action an OpenFlow switch will take upon receiving a packet whose header does not match any flow entry is to send it to the controller. So, when a switch receives a new packet, it will trigger a *Packet In* event. Our application will then run the respective event handler. It installs flow entries on the relevant switches (by sending a *Flow Modification* messages), triggering the switches to send the packet on its path until it reaches its destination. Subsequent packets belonging to the same connection will match on these flow table entries installed. No controller action is therefore needed (as long as packets keep matching on flow entries). These steps will be reviewed with greater detail in subsequent sections.

When the application needs to request flow table information, it instructs the controller to send a *Statistics Request* message to the switches. When it receives the reply from the switch, the controller will trigger a *Flow Statistics Received* event. The load balancer, having registered to listen for this event, collects the information it requested. We will later see how our load balancer application uses this information.

Connection Up events and *Set Configuration* messages occur in the startup stage of the application only. A switch can be configured to send along a specific number of the first bytes of an unmatched packet when communicating this occurrence to the controller. We use Set Configuration to configure the switches to send the whole packet. While this exerts additional overhead in some scenarios, it is negligible in ours.

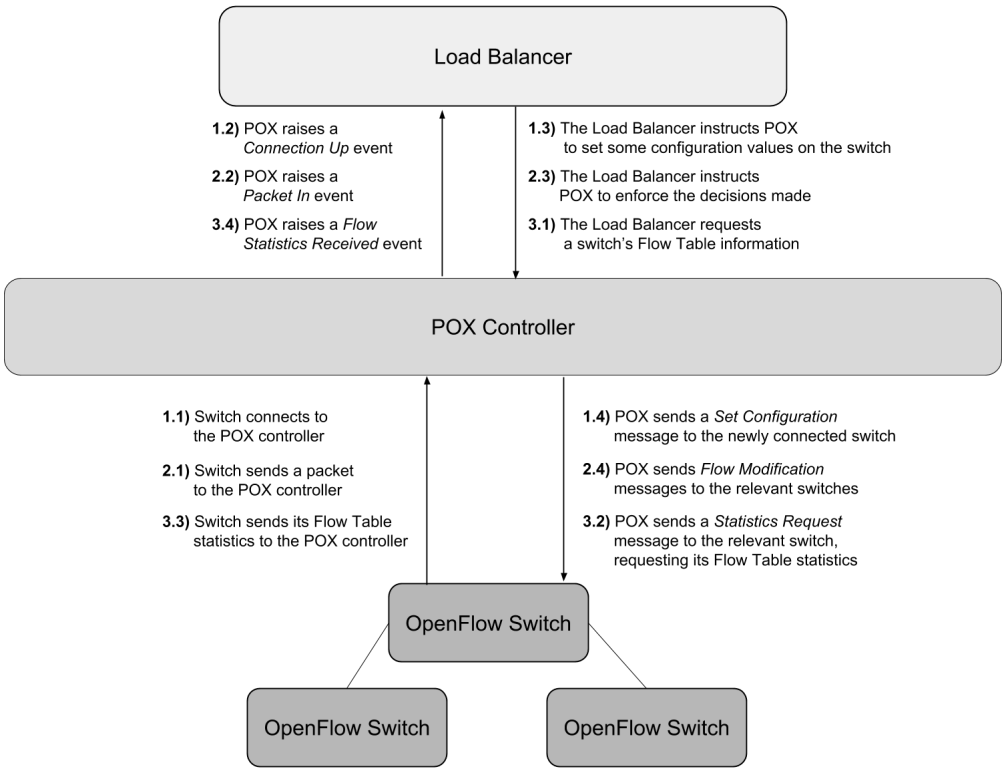


Figure 3.1: Communication between controller, switches and the load balancer.

A view of the communication between switches, controller and load balancer application, as these various events and messages are used, can be seen in Figure 3.1. We note that the three occurrences depicted in this figure are not presented in any temporal order. For example, while it is necessary for a switch to connect to the controller in order to be recognized and used by the application (messages 1.x in the figure), there is no relation between the arrival of new packets and the request for switch information (i.e., messages 2.x and 3.x can occur in different orders).

3.1.1 Flows

Flow entries installed in the switches must be made so that all packets from one connection match on them. This is due to the fact that several web services require servers to use stored client information in order to effectively service a request. This makes it imperative that all packets pertaining to a request be sent to the same server. For our purposes, we define connection as a TCP connection. As explained in the previous section, we can build flow entries that match on a single field (all other fields being wild cards), or on a set of fields. A TCP connection can be characterized by the 4-tuple $\langle$source IP address, destination IP address, source TCP port, destination TCP port$\rangle$. When creating flows, we construct them so that packets match on these four fields.

Flows have a time-to-live period. They can be configured to expire when we wish them to do so. Hard and soft timeouts can be attributed to a flow entry. A hard timeout will trigger a switch to delete the flow entry from its flow table after an imposed amount time has passed. A soft timeout will have a flow entry deleted only when the flow has been inactive for a certain amount of time. Inactivity in a flow entry means having no packets matching on it. On our setup, flow entries pertaining client-server communication have a soft timeout of 30 seconds, and have no hard timeout.

Each flow entry is associated with an action. In our setting we need to consider two. First, to route packets to their destination, the action the switches take is to *forward* the received packet to the egress port leading to the next hop in the path. Second, to translate from the virtual address to the effective server address that will treat the request (recall Section 2.2) we need to *change certain fields* of a packet, as a server host machine will only acknowledge a packet to be destined to it when its MAC and IP addresses are the same as the packet's destination MAC and IP addresses. Since clients do not know which server will service their request, the load balancer must write the target server's MAC/IP on these fields. The destination MAC/IP addresses of packets will then be the same as those of the chosen server.

3.1.2 Routing

As said before, the POX controller provides management applications with a network view. This dynamic view keeps changing as network events occur, and is a result of the controller's observations. In a typical network, control applications would be aware of topology changes, graciously accommodating nodes entering and leaving the network. Routing paths would be calculated dynamically, commonly by a routing application running on the controller. In such a scenario, our load balancer would use this dynamic routing information when building up the path choices for a resource. However, since routing is not in the scope of our work, we have opted to define the network topology and routing in a static way. The network topology remains unchanged throughout the experiments, and so do the available routing paths.

In our implementation, topology information is read from a file. This file provides our application with knowledge on server and switch's names, IP and MAC addresses, and switch's Data Path Identifiers (DPIDs) (a unique switch identifier). The names are useful to identify each node (switch or server) when running the application.

The file also provides the load balancing application with routing information. The application reads from the file details about every switch's ports, and to which nodes they lead to. This is what allows the load balancer to construct the available paths between nodes, and present the available choices when choosing a path to a server.

3.2 Load Balancing

After presenting the generic details of our application, we now proceed to explain how it distributes client's requests among the servers.

Each of the replicated servers has its own IP address in the network. However, the collection of servers is viewed by the client as one single entity that provides some web service. In our case, we assume the service to be a traditional web server, in which clients issue HyperText Transfer Protocol (HTTP) requests and the server sends a response. This web service responds to a single IP — an IP alias — and clients seeking to use the service will send requests to this IP.

When a client sends a request to the web service's IP alias, the switch connected to the client will raise a *Packet In* event on the controller if the respective packet header information does not match on any of the switch's flow entries. As explained before, this event is triggered by the request's first packet, and the load balancer handles it. The load balancer will then choose a server from the collection of servers using a particular scheduling policy. In certain cases it will also choose a specific path for the server. After the decision is made, the controller then installs flow entries on each switch's flow table down the chosen path. When the server response's first packet arrives at the switch connected to the server, it triggers another *Packet In* event, and the load balancer installs flow table entries for the reverse path of the reply packets in the flow table.

3.2.1 Handling Packet In Events

A switch raises a *Packet In* event in three distinct situations:

1. When it receives the first packet of a request issued by a client — A packet is ruled to be the first of a client-initiated HTTP request if the load balancer does not have a ⟨client IP, client TCP port⟩ pair stored for the packet header on its state. In this circumstance, the load balancer chooses a server and a path to the server, and stores this information in its state. How the load balancer makes this choice is explained in Section 3.3. Finally, a flow entry is installed in the flow table of every switch of the path. On the ingress switch that receives client traffic, the flow entry installed will not only enforce the action of forwarding the packet to the port leading to the next hop on the path, but will also rewrite the packet's destination IP and MAC addresses to that of the chosen server. Subsequent flow entries (on all other switches) will need only to route the packet down its path; no field rewriting is necessary.

2. When the first packet of a server reply is received — When a client request reaches a web server, the server will in turn issue a response. The response packets will not match on flow table entries previously installed for the client-to-server path. Just as

happens for the client-to-server communication path, the first packet of the server response will trigger a *Packet In* In event. This packet's fields will be the reverse of their respective client-to-server counterparts. If for the pair ⟨destination IP address, destination TCP port⟩ a ⟨server, path to server⟩ pair is stored as load balancer's state, the packet is from the server's response. Flow table entries are installed for the path retrieved from the load balancer's state, but in the reverse order. The first hop switch's flow entry will have an extra action to rewrite the source IP and MAC addresses to that of the web service's IP and MAC alias.

3. When the packet received is a packet pertaining to an ongoing connection that arrives at a switch before the respective flow table entry is installed — Flow entries are installed into the OpenFlow switches by the controller. However, there is no strict time guarantee as to when this happens. It is possible for a packet to reach a switch before the matching flow entry is installed. Because the switch does not yet know what to do with it, it triggers a *Packet In* event. As the load balancer already knows the route this packet will traverse (it has already made the decision, otherwise the packet would not be "in-flight") it can install the flow accordingly. Hence, when a packet arrives before its respective flow entry, the switch is instructed by the load balancer to forward the packet to the next hop in the path retrieved from the load balancer's state. Packets arriving at a switch before their respective flow entry is installed can occur either in the client-to-server path or the server-to-client path. Whichever the case, the load balancer determines the next hop, and instructs the switch to forward the packet to its correct egress port.

For our scenario, the load balancer's state is a data structure that, for each pair of client IP address and TCP port, stores the chosen server and the path to the server.

3.3 Scheduling Algorithms

We have explained the basic functionality of our load balancer application and how its decisions are enforced in the network. The scheduling decisions, which are at the heart of the load balancing behavior, will now be described.

Traditional load balancing solutions, as said before, balance load across server machines. These solutions are oblivious to the path the data takes to get to the chosen server. Therefore, they cannot take advantage of a network topology that allows multiple paths to a host machine. Scheduling load between multiple paths can be particularly beneficial when certain links become congested, or when a path's delay is higher than another. The sophisticated network control and management allowed in an SDN architecture makes it possible for our load balancer application to balance load not only across servers, but also choose the path the data takes to reach the chosen server.

In the novel algorithms we analyze, we choose both the server and the path to the server. We now explain the different algorithms and how they are implemented.

3.3.1 Server Choice Algorithms

These are the algorithms used to choose which server will service the request. The *Packet In* event handler will call one of these algorithms when such an event is raised, triggered by a switch that sees the first packet of a client-initiated request.

Round Robin

This is the simplest algorithm. Requests are attributed to each server one at a time, in a circular way. Due to its simplicity, it is also the lightest in regards to CPU usage. When a *Packet In* event is handled, triggered by the first packet of a client request, the chosen server is always the next on a list of all the available servers in the network. The order of the servers in the list is always the same. This leads to having all servers service a similar number of requests, independent of the load on each one.

Flow Connections

In this algorithm, the server with the least active connections is chosen. In case of a tie, the server with the lowest identifier is chosen. The load balancer retrieves the number of connections of a server by querying a switch for its flow table statistics. The switch queried must be a switch through which all traffic to the server must pass. Bearing in mind that flow entries are removed after thirty seconds of inactivity, by counting the number of flows leading to a certain server we can have a fair prediction on how many active connections exist in that server. The load balancer sends a *Statistics Request* OpenFlow message every Δ seconds and, when the respective switch responds, the controller raises a *Flow Statistics Received* event. The event handler counts the number of flows destined to a particular server in the switch, and the result for every server is stored. When scheduling load, the load balancer chooses the server with the lowest value of active connections, as per what is stored in its state. In our experiments we will vary the parameter Δ and observe how our load balancer performs.

Server Connections

As in *Flow Connections*, this algorithm chooses the server with the fewest active connections. Even in case of a tie, the outcome is similar — the server with the lowest identifier is chosen. The two algorithms differ in the method to retrieve the number of connections. In *Server Connections*, the servers themselves periodically send this information to the controller. The server machine runs a program that verifies how many connections it has using the command *netstat –tn*. This command outputs the machine's current TCP connections. The program then counts how many lines this command outputs, and sends a

User Datagram Protocol (UDP) packet destined to a "ghost IP" containing this information. The term *ghost* emphasizes that no machine on the network has that address as its own. The switch receives this packet and, since it does not have a matching flow entry for it, raises a *Packet In* event. Since it is addressed to a specific "ghost IP", the load balancer knows what it is and what to do with it. It reads the packet data to retrieve the number of active connections and associates it with the respective server. Servers send a UDP packet every Δ seconds, which means the load balancer has this information updated every Δ seconds.

Load

The server chosen in this algorithm is the least loaded server. We simplify the meaning of load, and compare only the server machine's current CPU load. Servers send their current CPU load, and the load balancer chooses the one with the smallest value. The server machines assess their current CPU load using the command *mpstat*. Just as is done for *Server Connections*, a UDP packet containing this information and addressed to some known "ghost IP" is sent by a program running on the server machine every Δ seconds. The load balancer retrieves the packet, and stores the CPU load values for each server. When asked to choose a server, it replies with the server that has the smallest CPU load value.

It should be noted that, when running Mininet Hi-Fi, the *mpstat* command will yield the total CPU load on the host machine, and not the specific usage of a server machine emulated in a Linux container. In order to obtain this value, we have to change the script that runs on the servers to collect this information. We use the command *cpuacct.usage_percpu*, which reports the CPU usage of a particular *cgroup*.

This algorithm, as with *Flow Connections* and *Server Connections*, does not produce responses that are guaranteed to be correct. In the time interval it takes for this relevant information to update, there is a risk of producing erroneous results. There is therefore a compromise between having the most up to date information and flooding the network with these "control" packets. Different time intervals will best fit different scenarios.

Response Time

In this algorithm, the server with the lowest perceived value of response time is chosen. The elapsed time between the arrivals of the client request's first packet to the switch connected to the server and the server response's first packet at that same switch is measured and stored. The server with the lowest average value across an average window of Δ seconds is chosen.

To make this possible, a simple change in the behavior of the load balancer is necessary. To have a clear idea of the response time, our application must know when the client request's first packet arrives at the switch connected to the server, thus "removing"

path delay. If the respective flow entry is installed before this packet reaches the switch, the load balancer has no way of knowing when that happens. Instead, we deliberately do not install a flow entry for the last switch on the client-to-server path. When this switch raises a *Packet In* event in result of this situation, the load balancer can now produce a timestamp that enables it to calculate an approximate response time.

Whenever the first packet of a new client request reaches the last switch of the client-to-server path, the load balancer stores a timestamp on its state. When the server response's first packet triggers a *Packet In* event on the ingress switch of the server-to-client path (which is the same switch as used in the previous timestamp), the load balancer generates another timestamp. The elapsed time between the two timestamps is stored. All response time values are stored during a window of Δ seconds to calculate the average. The load balancer chooses a server with the lowest average response time value.

3.3.2 Path Choice Algorithms

Following the choice of the server that will handle a client request, the load balancer has to choose the path to that server. When both the server and the path to the server are chosen, the load balancer can instruct the controller to install the respective rules in the data path.

Shortest Path

This algorithm will return the shortest path available between two nodes. The load balancer knows all the available paths from its stored network topology. The shortest path length is determined and all paths with that length value are isolated in a set. In this policy the chosen path is always the same. If the isolated set has more than one member, the first one is chosen. Since the order of the set never changes, the chosen path is always the same.

Equal-Cost Multi-Path

With Equal-Cost Multi-Path (ECMP) routers can decide where to forward packets when there are multiple choices of equal cost. The cost is calculated using some routing metric, and a destination port (a path) is chosen between those that tie for best value [20]. Networks employing the Open Shortest Path First (OSPF) routing algorithm often use ECMP to perform load balancing over their links.

In our scenario, *Equal-Cost Multi-Path* runs only once to determine the whole path, and the applied metric is the length (the number of hops). A set of all paths with the shortest available length is created. If there is only one shortest path, that one is chosen. In case multiple paths tie for shortest, a round-robin scheme is used to select between them.

Path Delay

Given a set of paths available to a server, this algorithm will choose the one with the lowest path delay. Path delay is calculated by measuring the round trip time a packet takes from the switch connected to the client and the server.

For this purpose, the load balancer generates a UDP packet that travels across all paths to all servers. This is guaranteed by associating an identifier to a path (for convenience, we have used the UDP source port for this purpose); that is, the tuple $\langle$client, server, identifier$\rangle$ uniquely describes a path in the network. Flow entries are installed in all switches to guide these control packets through their respective paths. These UDP packets are generated with a "ghost IP" as the source IP address, so that load balancer and servers alike know what to do with it. The servers are equipped with a script that, whenever they receive one of these "ghost IP"-sourced packets, they send a reply UDP packet to the same "ghost IP". When the reply reaches the switch connected to the client, the load balancer calculates the elapsed time taken for the round trip the packets made. To eliminate server processing time from this path delay, the controller calculates two elapsed times. The elapsed time the first UDP packet takes between the first and last switches on the client-to-server path, and the elapsed time the response UDP packet takes between the first and last switches of the server-to-client path are registered. The effective path delay, without the server processing time, is the sum between these two registered elapsed times.

This is done every Δ seconds, which means that the round trip times in each path are updated every Δ seconds. The values collected are registered over an average window of 60 seconds. When the load balancer needs to decide on a path to a server, it simply chooses the one with the lowest average path delay.

3.4 Final Remarks

In this chapter we have seen how our load balancing solution achieves its desired result. We have shown how, by instructing switches on how to handle new flows, load balancing behavior is achieved in a Software-Defined Network. As the load balancer has full control over the path the traffic takes, it can instruct switches to schedule incoming connections between the available replicated servers and the multiple paths to a server. We have also described the scheduling algorithms implemented. When handling a new connection, the load balancer must decide on which server to service the request, and the path taken by the traffic to the chosen server. To that effect, we have implemented a number of both server choice and path choice algorithms.

In the next chapter we will evaluate the feasibility of our load balancing solution, and compare the performance of the implemented algorithms.

Chapter 4

Evaluation

In this chapter we provide details regarding the experiments done to evaluate our SDN-based load balancer. The purpose of this evaluation is to ascertain the functionality of our application, and to compare the various load balancing algorithms implemented.

First we give a description of the network topology used. We will then show how Mininet Hi-Fi is used and how the load balancer application is run. Finally, we will provide details of the experiment and analyze its results.

4.1 Network Topology

The topology used for the experiments is based on the typical multi-rooted, layered topology used in data centers today [27]. Such a topology is used to maximize system throughput and minimize network latency using commodity Ethernet switches [1]. In it, switches are grouped in layers, allowing for multiple paths between nodes while maintaining a tree structure. Typically, servers are placed in a rack, and are connected to a Top of Rack (ToR) switch (also called access switch). These connect to another layer of switches, called the aggregation layer. At the top of the hierarchy, core switches interconnect the whole network, managing inbound and outbound traffic. We note that this is the network architecture used in most datacenters, as attested by the extensive work done in similar topologies [1, 2, 14, 17, 27].

In our topology, switches are divided in three layers — the core, aggregation and access layers. The server machines connect to the bottom (access) layer of switches, while the clients connect to a switch on the top (core) layer. The core switch connects to all the aggregation switches, and each aggregation switch connects to all the access switches. All links have the same bandwidth of 1000 Mb/s, as in a typical datacenter. A view of the network topology can be seen in Figure 4.1.

We consider a simple network with five switches. The aggregation and access layers consist of two switches each. There is a single switch in the core layer – the core switch. We have four servers, with each access switch connecting to two servers. Two clients are

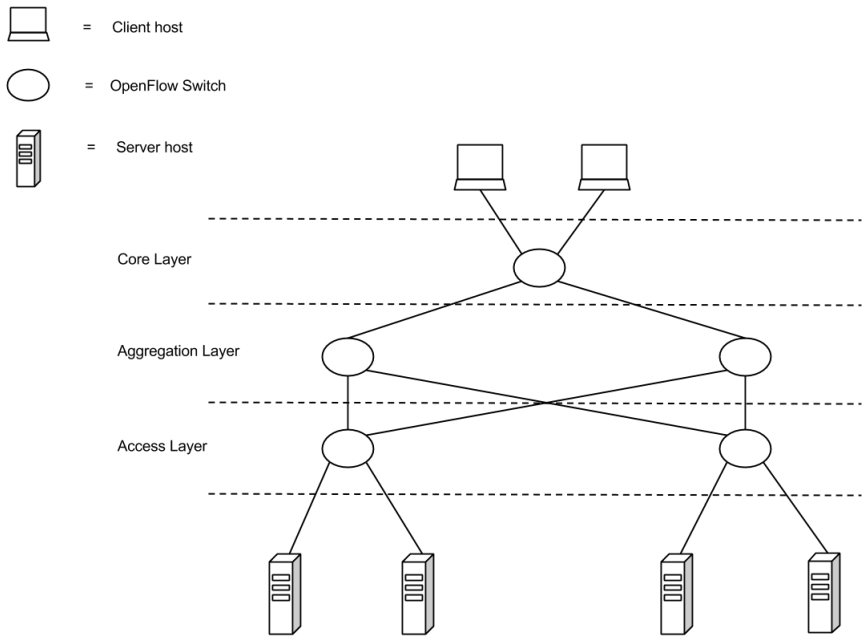


Figure 4.1: A view of the experiment's network topology.

connected at the core switch.

4.2 Mininet Hi-Fi Setup

We have performed our experiments using Mininet Hi-Fi. As explained before, Mininet Hi-Fi is an emulator that allows for the rapid prototyping of Software-Defined Networks. It provides an environment with a network of interconnected virtual switches and hosts capable of running real kernel, switch and application code. We can create a network based on our needs, and exert on it the behavior we desire.

Mininet Hi-Fi employs software-emulated OpenFlow switches that provide the same semantics of a hardware switch. The controller can be located either on the real or on the emulated network [23]. The controller's IP address and listening port is provided to Mininet Hi-Fi when it starts. In our setup, POX runs on the same machine as Mininet Hi-Fi. When we run Mininet Hi-Fi, we configure it so that the virtual switches connect to a controller in the localhost (IP $127.0.0.1$) and port 6633.

We have created a separate Python file from which Mininet Hi-Fi reads the topology used. Names and addresses of our network nodes are defined in this file, as are the network links between them. All network characteristics are defined by manipulating Mininet Hi-Fi's Python objects, using its API. We also specify the properties that aim to provide performance fidelity, namely link parameters and host CPU limits.

The same Python file is used by our load balancer application to retrieve the relevant network topology information it needs to work, as described in Section 3.1.2.

4.2.1 Virtual Links

Virtual links are also defined when creating the network topology in the Python file. As said before, the bandwidth of all links is 1000 Mb/s.

To emulate a slower path, one link is configured with a delay value of 600 milliseconds, while others have no delay value specifically configured. We chose this value to clearly assess the performance of the different load balancing algorithms.

4.2.2 Virtual Hosts

In section 2.3.1, we have seen that virtual hosts are separate containers that provide processes with their own network namespace. This means that each container has ownership of exclusive interfaces, ports, and routing tables [23].

To guarantee performance fidelity, as explained in Section 2.3.2, we must split the CPU among the containers while leaving some margin for packet forwarding. Also, we have to monitor the experiment to ensure it is running within the limits imposed and yielding realistic results. This means that there is not a conventional, established configuration; we must infer the CPU demands of packet forwarding and of our virtual hosts. As seen in [18], ensuring the presence of idle CPU verifies that the experience has not fallen behind an ideal execution schedule on hardware.

Empirical knowledge gained by running a number of experiments using our load balancer allowed us to conclude that the CPU usage for packet forwarding is significantly inferior to the virtual host's demands. This is due to the fact that path length, lookup complexity and link load are less significant when compared to the server's computations. As a result, we decided to allocate 90% of our CPU load to all the hosts and leave 10% for packet forwarding. Each of the four servers has a CPU slice of 18% and each of the two clients has a CPU slice of 9%.

4.3 Experiment Setup

4.3.1 Client

The client program performs HTTP requests to the web service's IP alias (10.0.0.100). The client then monitors the elapsed time between sending the request and getting a reply, and logs that information into a file. The log contains a timestamp for the request, the server that replied to it, and the response time.

The client is configured to have two concurrent threads that send eight requests in a row (thus, a total of sixteen requests per run, per client). Client requests trigger a Common Gateway Interface (CGI) script in the server.

Our experimental network setup has two client machine hosts connected to the core switch, as seen in Figure 4.1.

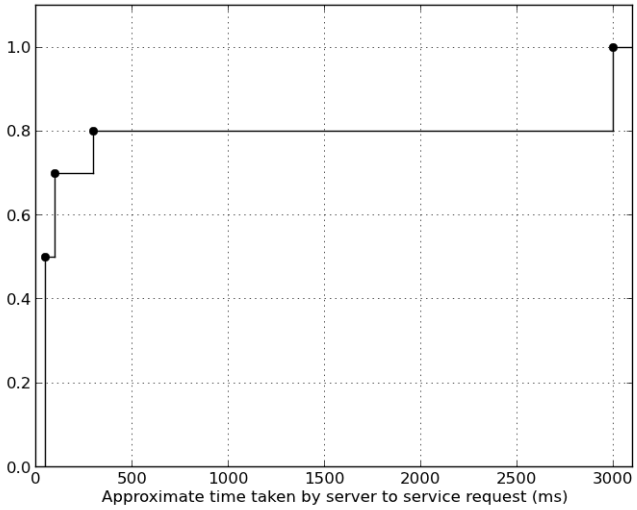


Figure 4.2: CDF for the base request service load distribution function.

4.3.2 Server

A simple server program runs on all server (virtual) machines. It executes a CGI script per client request. This script returns information on the server that handled the request and the elapsed time taken to do so.

The script is also made to generate different CPU loads on the server. Using a random number generator, requests generate random loads on the server machine. In the first set of experiments, an average of 20% of requests handled incur a heavy load on the CPU, causing a delay of approximately 3 seconds for the server to handle the response; around 10% of the requests cause a delay of 300 milliseconds; around 20% 100 milliseconds; and the remaining 50% of requests are handled without any "artificial" delay. A Cummulative Distribution Function (CDF) of this distribution can be seen in Figure 4.2.

4.3.3 Procedure

Our goal is to evaluate the performance of the different load balancing algorithms. The performance metric we use is the response time experienced by the client. The response time is a good measure of the system performance: the quicker requests get serviced, the better is the client perceived system's performance. Generally, an algorithm achieving lower average response time implies that it is utilizing the cluster resources well and balancing loads among the server nodes fairly [36].

A run of our experiment is characterized by having the two client hosts concurrently running the client program twelve consecutive times. As explained before, the client program sends sixteen HTTP requests. This means that for each run of the experiment, the two clients generate $12 \times 16 \times 2 = 384$ HTTP requests.

Our load balancer will be used to evaluate several scheduling algorithms, based on two dimensions: algorithms that choose the server and algorithms that choose the path taken by the client to the chosen server. The load balancer will thus execute two algorithms when scheduling a request.

Server Choice Algorithms	Path Choice Algorithms
Round Robin	Shortest Path
Flow Connections	Equal-Cost Multi-Path
Server Connections	Path Delay
Load	
Response Time	

Table 4.1: The server choice and path choice scheduling algorithms.

	Equal-Cost Multi-Path	**Shortest Path**	**Path Delay**
Round Robin	RR-ECMP	RR-SP	RR-PD
Flow Connections	FC-ECMP	FC-SP	FC-PD
Server Connections	SC-ECMP	SC-SP	SC-PD
Load	L-ECMP	L-SP	L-PD
Response Time	RT-ECMP	RT-SP	RT-PD

Table 4.2: The scheduling algorithm combinations formed by server choice algorithms (rows) and path choice algorithms (columns).

The eight algorithms (five to choose the server, three to choose the path) are shown in Table 4.1. We execute our tests on all combinations of ⟨server choice, path choice algorithms⟩, a total of fifteen different pairs to test. A view of all combinations of algorithms can be seen in Table 4.2. We will use the nomenclature seen in these tables to refer to the various algorithms.

For each of the fifteen combinations of algorithms, ten runs of the experiment are executed. All 384×10 response time values are used as data to analyze the performance of our load balancer.

As we have seen in Section 3.3, some scheduling algorithms have configurable parameters. The Δ parameter has different meanings for different algorithms, but it always represents a time interval. In this experiment, all algorithms are configured with $\Delta = 5$ seconds.

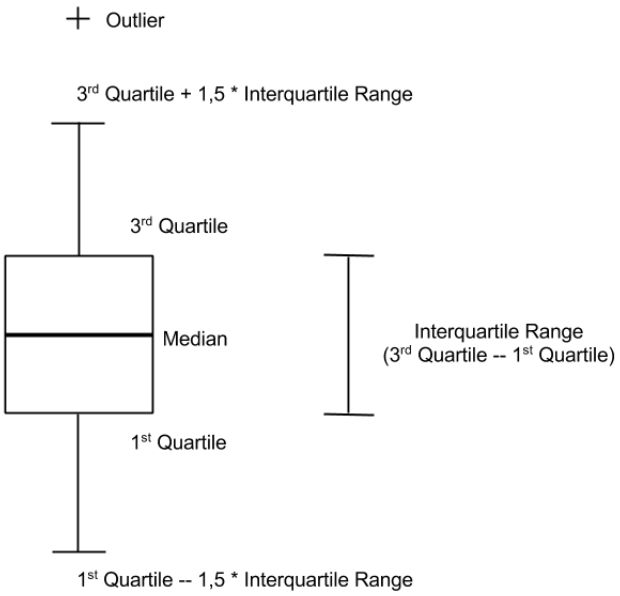


Figure 4.3: Pictoral depiction of how to read data from a boxplot.

4.3.4 Verifying Fidelity

Mininet Hi-Fi is a network emulator that attempts to provide timing realism characteristics. As explained in Section 2.3.2, this is achieved by allocating and limiting CPU and link bandwidth, and monitoring the experiment.

For experiments that rely on accurate link emulation, we must monitor the dequeue times on every link and compare them to those of an ideal link. Examples of such experiments and how they were monitored can be seen in [18].

Our scenario, however, does not rely on accurate link emulation. It depends on a coarse-grained metric, the response time felt by the clients. All we need to verify is that no virtual host is starved for CPU resources and that the system has enough capacity to sustain the network demand [18]. As described in the previous section, we need to measure CPU idle time during the experiment. In all our runs, the system had at least 45% idle CPU time on every second. From this measure we can conclude that the system was able to schedule all virtual hosts and data transmissions without losing fidelity.

4.4 Results

In this section we present an evaluation of the performance of the different load balancing algorithms using our load balancer control application.

All response time values are condensed into boxplot diagrams. Boxplots graphically summarize groups of numerical data, in a pictorial depiction of how dispersed and skewed

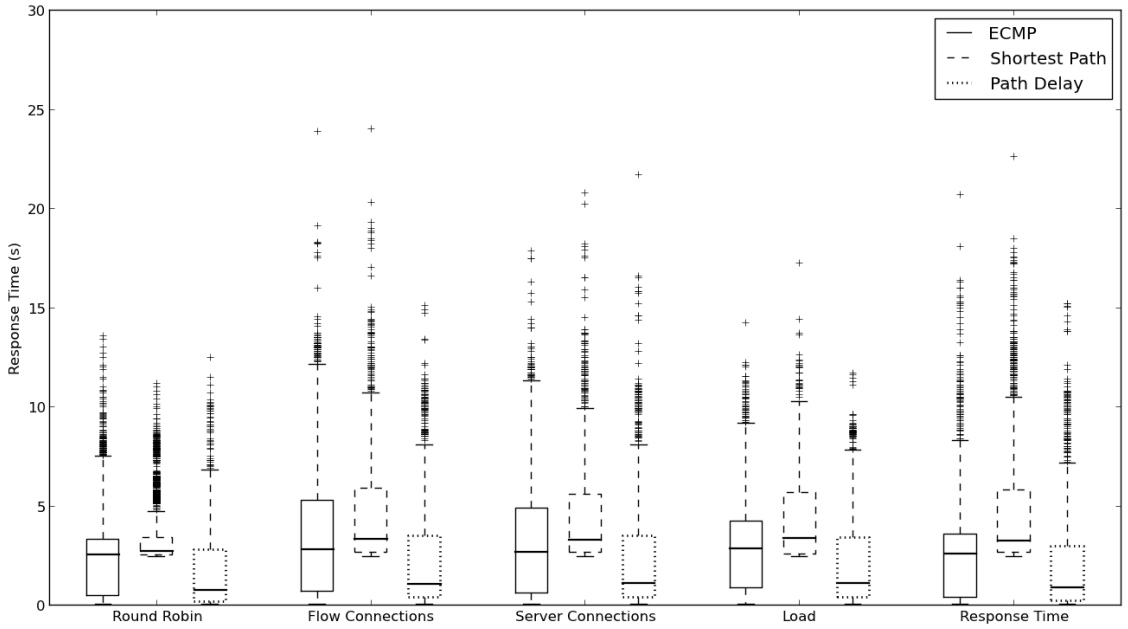


Figure 4.4: Boxplot of the response times for the experiment with default configuration.

the data is. Boxes are delimited by the data's 75th and 25th percentiles (third and first quartiles), respectively. Dashed lines — the whiskers — extend to the most extreme data points within a range defined by the interquartile range (third quartile minus the first quartile) multiplied by 1.5. Values outside this range are considered outliers, and are represented as crosshairs. A visual explanation can be seen in Figure 4.3.

The results of our experiment are shown in Figure 4.4. This experiment considers only the default parameters explained before. The main thing we can observe is the effectiveness of scheduling the load between available paths. Whichever server choice algorithm is used, an improvement over the mean response time is seen when using a path choice algorithm that schedules load between multiple paths (as explained in section 3.3, the *Equal-Cost Multi-Path* and *Path Delay* algorithms schedule traffic between paths, whereas in this network topology the *Shortest Path* algorithm always chooses the same path). Both *Equal-Cost Multi-Path* and *Path Delay* provide better performance than *Shortest Path*, whichever server choice algorithm is used. The highest performance improvement can be seen to occur when using *Path Delay*. Even though *Equal-Cost Multi-Path* also schedules traffic between available paths, we can observe that the performance gain when using *Path Delay* is significant. This allows us to conclude that algorithms that take network state into account when choosing a path perform better than those that make scheduling decisions independent of the state of the network. This is especially true when the delay of the available paths is significantly different. In fact, as seen in [31], there is a high

degree in path delay variability in the internet. Experiences conducted in [35] show that changes within the routes (e.g., load variability) account for approximately 70% of the delays in the internet. Thus, having a mechanism to account for such unpredictability is certainly beneficial.

Next we compare the server choice algorithms. One thing we can observe is that *Round Robin* slightly outperforms every other server choice algorithm. Even though when comparing the medians there is not much of a difference, it has less variance when compared with other algorithms. This is likely due to *Round Robin* being significantly lighter in terms of CPU usage when comparing to the other algorithms, a conclusion that can also be seen in [4].

4.4.1 Changing Update Intervals

We have evaluated various scheduling algorithms used by our load balancer application to partition traffic among available servers and paths. Some algorithms use a specific update interval Δ, either for an averaging window, or to define when to update some metric of interest.

- *Flow Connections*, *Server Connections* and *Load* — the update interval dictates how often the metric information is updated.
- *Response Time* — the time value of the average window.
- *Path Delay* — how often the controller sends control packets that allow it to update the delay values for each path in the network.

To further evaluate our load balancing solutions, we ran the experiment depicted in section 4.3.3 with different configurations for each algorithm. The results will be divided into three sections. Algorithms *Response Time* and *Path Delay* are studied separately. Since *Flow Connections*, *Server Connections* and *Load* are similar in their procedure and configuration scheme, we have grouped them together, configuring and presenting their results as a group in a separate section.

Path Delay

In this experiment we varied *Path Delay*'s update interval from 2 seconds to 20 seconds. When using a low value such as 2 seconds, we aim to see how the service performs when we stress the network with control packets pertaining to the *Path Delay* algorithm, for maintaining an updated view of the network. Likewise, when we increase the delay to 20 seconds, we want to observe how the system performs with minimal overhead on the network, and potentially stale information.

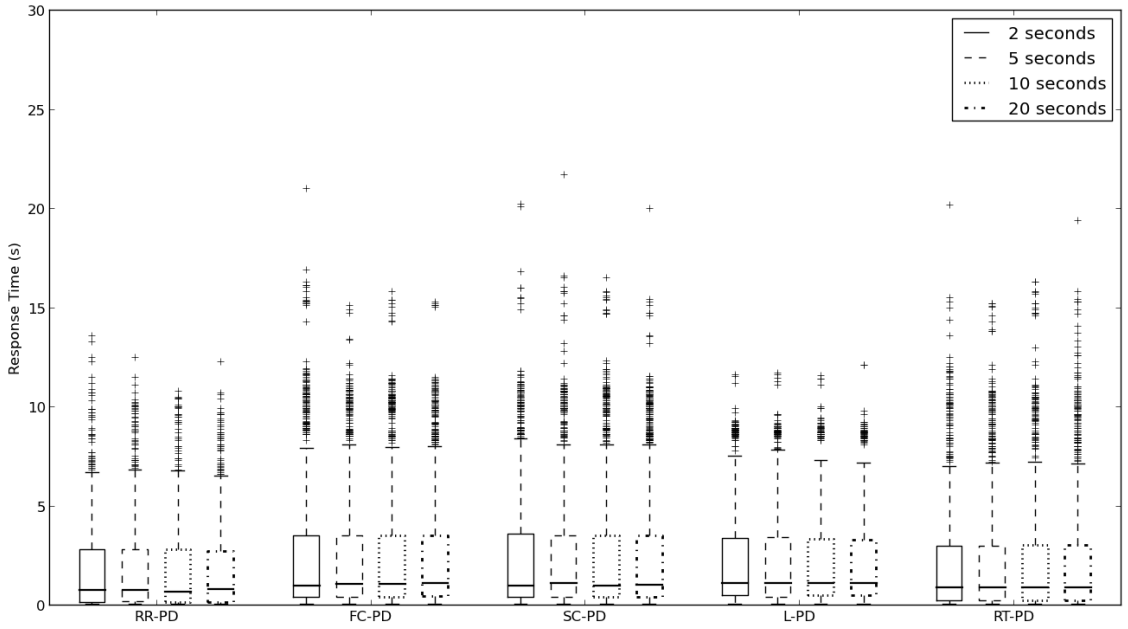


Figure 4.5: Response time for different configurations of Path Delay.

We also tested values under 2 seconds, but this caused an excessive strain on both the network and the controller, which resulted in unpredictable behavior.

Since we only changed *Path Delay*, any differences will be the result of changes to this policy. We present the results in Figure 4.5. We can see there is no change in performance when we alter the value of *Path Delay*'s update interval. Even when we increase the update interval to 20 seconds, the performance remains similar. One possible reason for the performance to remain unaltered is that the network load is static. Since HTTP requests clients make are not for fetching any files, server replies are always the same size. This results in static network loads, which in turn results in unvarying response times.

Since there is no change in the algorithm's performance, there is no benefit in having a small update value. Using a less stringent update value performs as well as the other scenarios, with the added benefit of causing fewer load on both the network and the controller.

Response Time

Now we will test how changing the duration of the average window affects the *Response Time* algorithm. The average window is the period of time during which all response time values are averaged.

We run the experiment with average window values varying from 0.5 to 20 seconds. Unlike all other algorithms, these different values do not incur additional stress on the

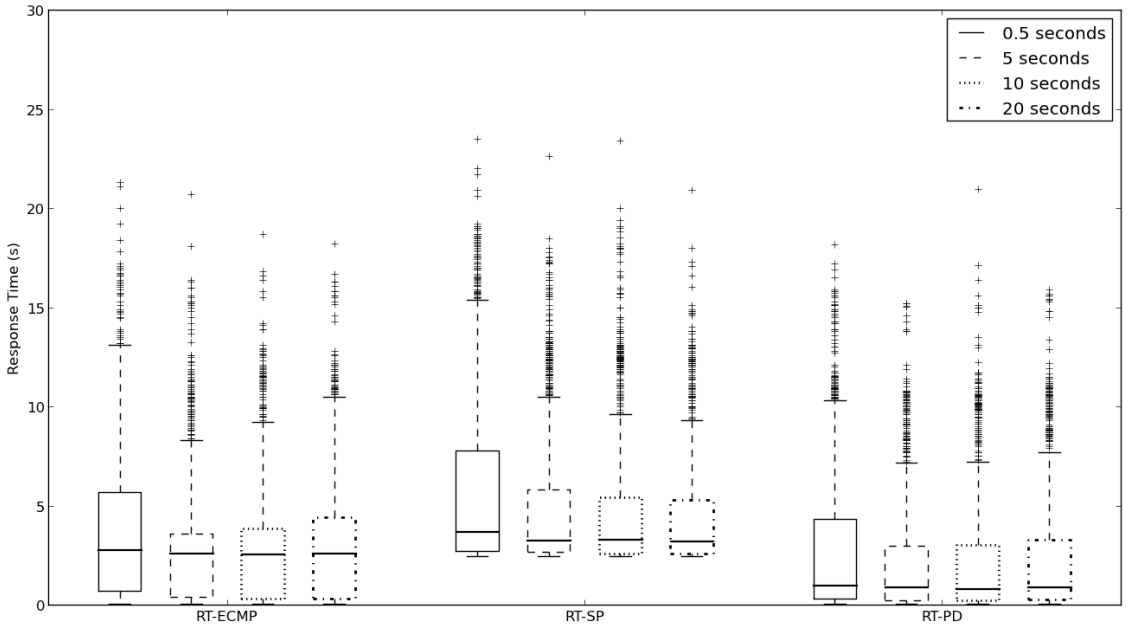


Figure 4.6: Response time for different configurations of Server Choice Response Time.

network. The load balancer application obtains the response time values used in this algorithm by monitoring the actual traffic between clients and servers. Having a "wider" average window (with a bigger time interval) only means that more values will be used to calculate the average.

The results are presented in Figure 4.6. We can see that a decrease in the value of the average window adversely affects the performance of *Response Time*. There is a higher variance in the client perceived response time when using this server choice algorithm with an average window of 0.5 seconds. Having such small averaging window means that the controller is not able to keep track of the response time history, which is prejudicial to the overall performance. On the other hand, increasing the average window from 5 to 20 seconds has shown no benefits.

Flow Connections, Server Connections and Load

These three server choice algorithms have a similar configuration and for that reason we present their results together. We have run the experiment using update intervals from 0.1 to 10 seconds. As was the case for *Path Delay*, we chose these values to emulate "extreme" scenarios, where we incur a big overhead on the network aiming to achieve better performance, and gradually moving to others where the overhead is minimal.

The results of the experiments can be seen in Figure 4.7. There is an improvement when using smaller values of update intervals of under 1 second over the remaining, as

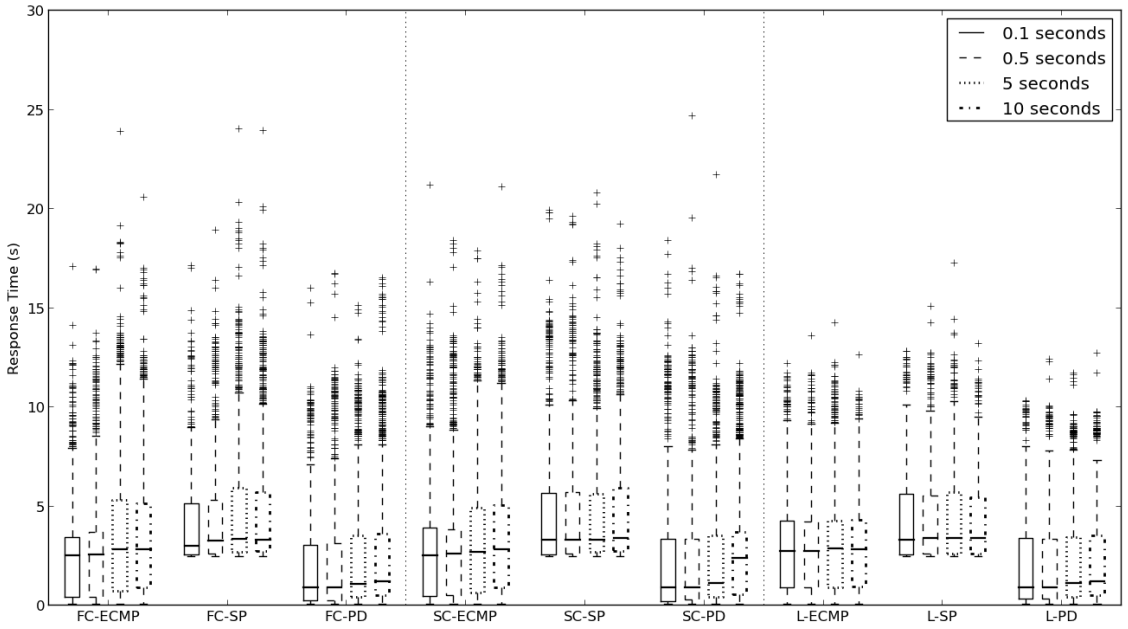


Figure 4.7: Response time values for different configurations of *Flow Connections*, *Server Connections* and *Load*.

is seen by the slight decrease in these algorithms' response time, not only in the median but also in variability. Performance is increased when the update interval decreases to 0.5 seconds and less; despite exerting more overhead on the network.

When using an update interval of 10 seconds, the overhead imposed on the network is minimal in comparison with all lower values; however, the performance also decreases. This can be observed in the slight increase in the median for all algorithms.

Optimal Configuration

We have changed various configuration parameters in our scheduling algorithms, aiming to discover in what setup they performed best. The conclusions we have reached in the previous section allow us to propose an "optimal" configuration for our system, configuring each algorithm individually. We aim to hit the "sweet spot" — to have good performance while striving to enforce minimal overhead on both the network and the controller.

Starting with path choice algorithms, as we have seen in the previous section, only *Path Delay* has configurable parameters. We can alter the time delay between the collection of each path's delay value, accomplished by registering the round trip time a packet takes. We have observed in Figure 4.5 that there is no performance difference between the various values studied. Given that performance-wise all configurations yield the same

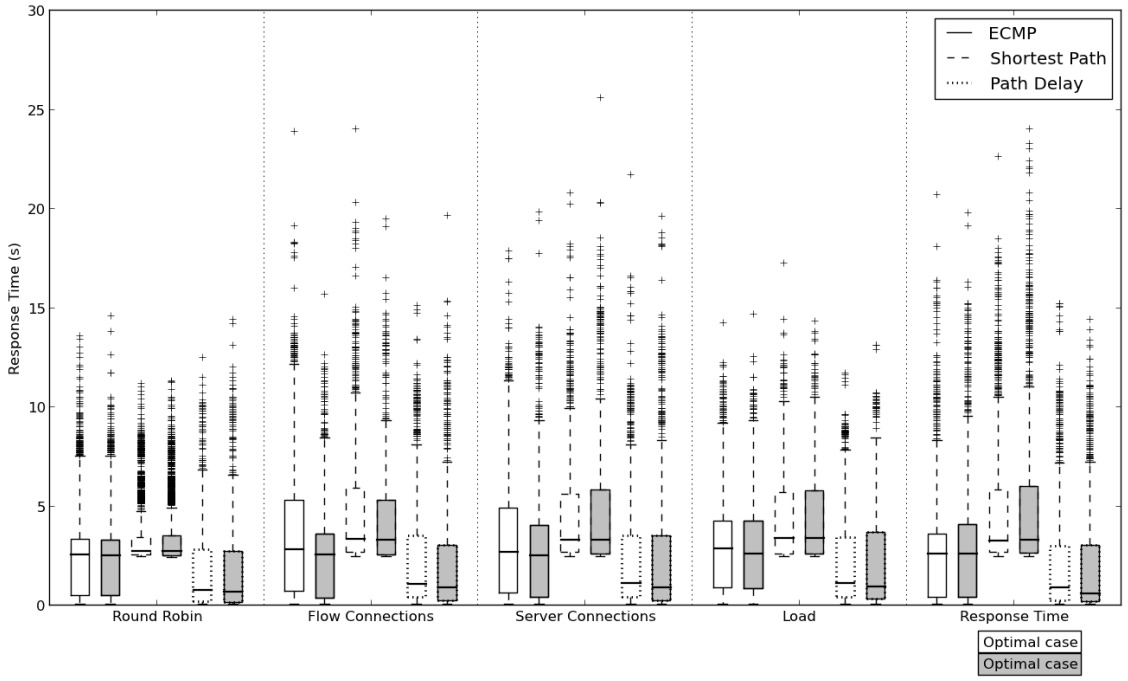


Figure 4.8: Boxplot of the response times using optimal algorithm configurations.

results, the optimal configuration is the one that exerts the minimal overhead. When using the longest delay value (20 seconds), there is less stress on the network and controller. This will be the value used in our optimal configuration.

In *Response Time* we can configure the time interval for its averaging window, a window of time during which all registered values are averaged to determine which server will service a request. By taking a look at Figure 4.6 we can see that, even though the performance is similar with an average window of 5 and 10 seconds, using the latter provides a slight advantage when using *RT-PD*. We chose this value, 10 seconds, as our optimal value for the *Response Time* average window.

The remaining server choice algorithms, *Flow Connections*, *Server Connections* and *Load* are configured in a similar matter: we can alter the update interval of their respective metric. Due to this similarity, we were able to configure them as a group. The results were condensed in Figure 4.7. We were able to see that these algorithms perform best when using small update intervals. However, small update intervals incur additional overhead on the network and the controller. There is no significant improvement when using an update interval of 0.1 seconds over using an update interval of 0.5 seconds. Since the latter exerts less overhead on the system than the former, we defined an update interval of 0.5 seconds as optimal.

We ran the experiment using these optimal configuration values. The results of this experiment were put side by side to compare with the first experiment (shown in Figure 4.4)

and can be seen in Figure 4.8. When comparing to the base case (the white boxplots), we can see there was an overall slight improvement on the response time felt by the clients, namely when using *Path Delay*. To have a quantifiable idea of the improvement, the median response time dropped by 9% for *RR-PD* (the least significant decrease), and by 32% in the *RT-PD* case (the most significant percentual improvement). The performance improvement in algorithms *Flow Connections* and *Server Connections* can be seen not only in the median but also in the third quartile. Even though we can see an improvement in *Response Time*, algorithm *Round Robin* is still performing better than all the others.

4.4.2 Changing Service Time Distribution

The CGI script that runs on the web servers every time these service a request randomly exerts additional load on its machine's CPU for different requests.

To understand how the client request patterns affect our conclusions, we evaluated all algorithms with different distribution for the request service load. We ran the experiments with a heavy-tailed and a discrete uniform distribution.

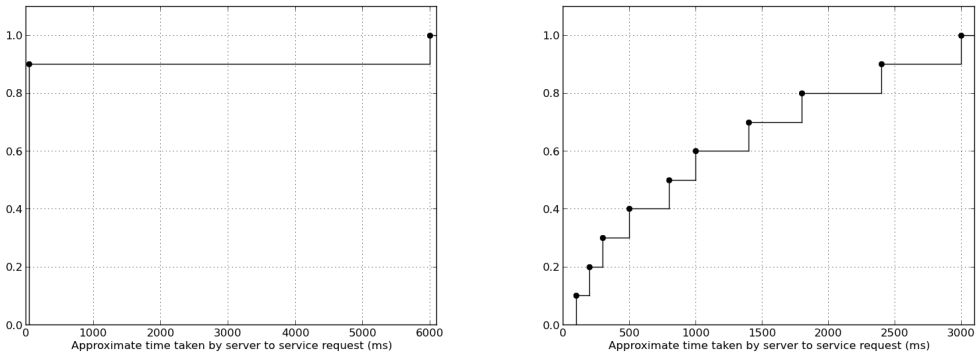
Heavy-tailed distributions have been observed in the internet in the context of traffic characterization [9]. They are observed in a wide range of computer workloads, and can be used to characterize web file sizes and the number of clients accessing a web service [6, 10]. A heavy-tailed distribution is one where there is a large probability of getting very small values, and a small probability of getting very large values. In our case, there is a 10% probability of the request incurring a load on the server which causes an approximate delay of 6 seconds. The remaining 90% are handled normally. A CDF of this heavy-tailed distribution can be seen in Figure 4.9(a).

In our discrete uniform distribution, several increasing load values are exerted in the server, each with the same probability of 10%. We can see the CDF for our discrete uniform distribution in Figure 4.9(b).

We ran the experiment described in Section 4.3.3, with the Δs for each algorithm configured as was for the optimal case. This means all results in this section will be compared to the results shown in grey, in Figure 4.8.

Heavy-Tailed Distribution

The results of our experiment running with a heavy-tailed service load distribution can be seen in Figure 4.10. We can observe that the variability decreases in the response time values experienced by the client. We conjecture this to be due to the higher number of requests (90%) that exert no additional load, hence decreasing variability. Interestingly, when considering *Load*, the variation increases. This is probably due to the 10% of requests that exert a very high CPU load. Recall that in the base case — Figure 4.2 — the maximum service request time was equal to 3 seconds, whereas in the heavy-tail scenario this time doubles.



(a) CDF for the heavy-tailed request service load distribution function

(b) CDF for the discrete uniform request service load distribution function

Figure 4.9: CDFs for the heavy-tailed and continuous distributions.

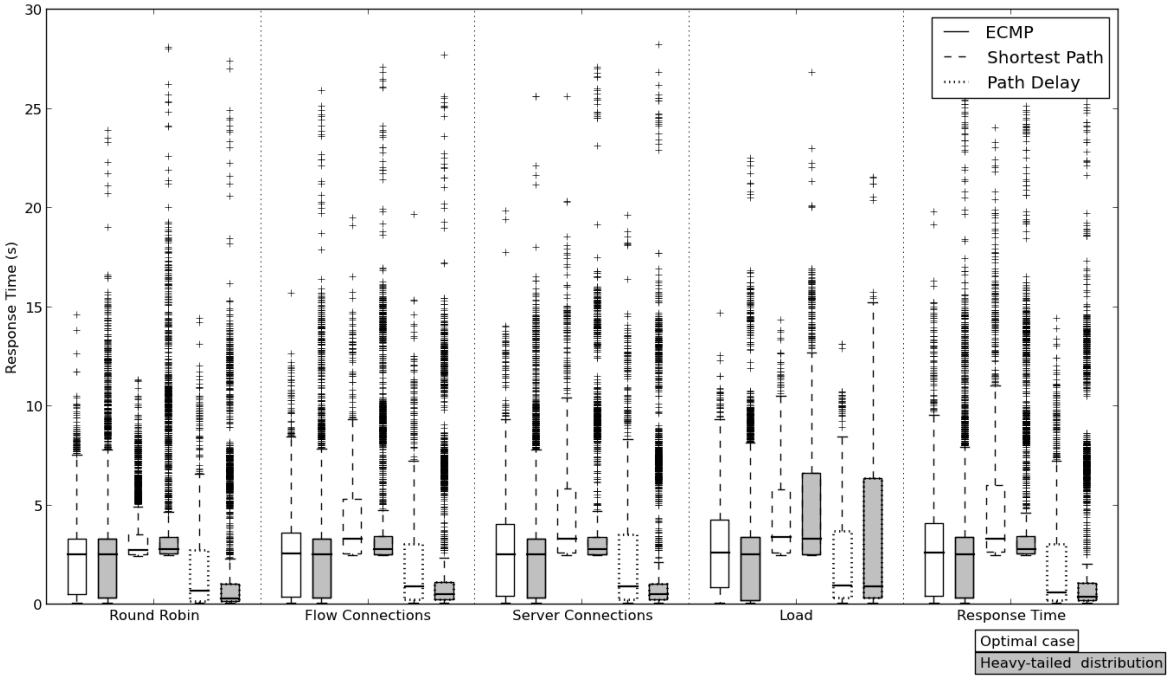


Figure 4.10: Boxplot of the response times using a heavy-tailed request service load distribution function.

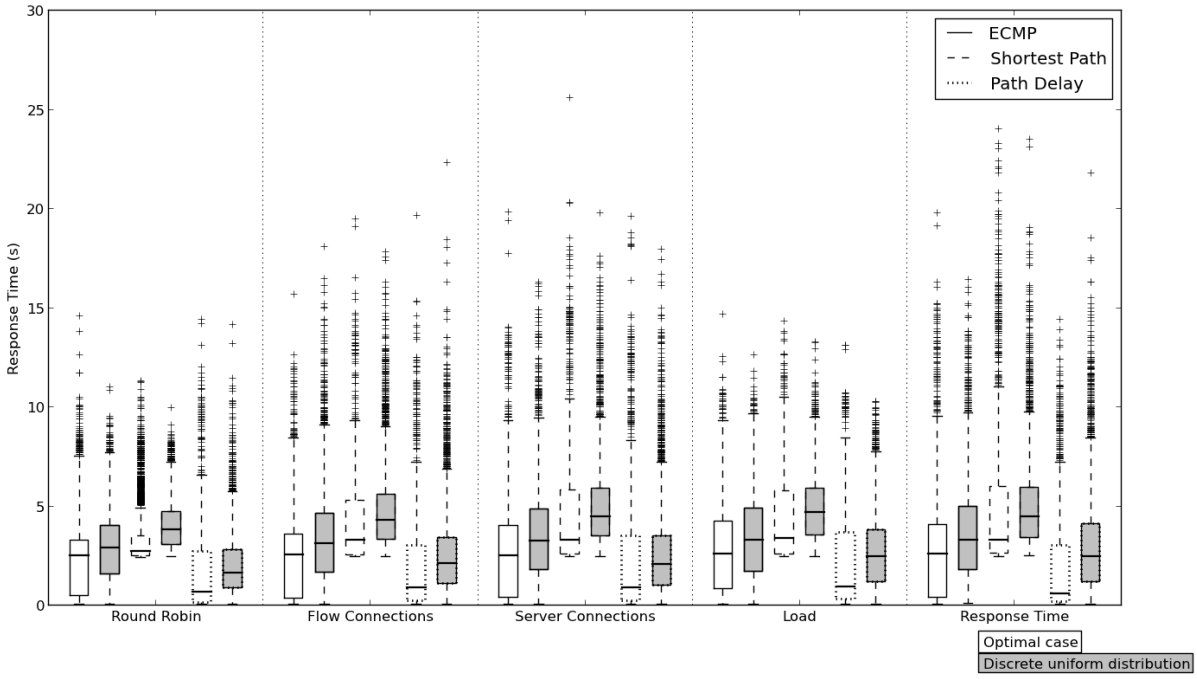


Figure 4.11: Boxplot of the response times using a continuous request service load distribution function.

Discrete Uniform Distribution

The results of running the experiment with our discrete uniform distribution are depicted in Figure 4.11. We can see that there is a clear reduction in performance. In all cases, there is an increase in the response time experienced by the client. This can be explained by the higher number of requests that exert high loads on the server. For instance, in the base case, only 20% of the requests have a service time above 300 milliseconds. On the contrary, in this scenario, 80% of the requests have a service time above that number.

4.5 Discussion

This chapter presented the experimental evaluation performed with to our SDN load balancer. We have characterized the network topology we used to implement a web service of four replicated servers, with two clients performing HTTP requests and registering the elapsed time between sending a request and obtaining a reply. Using Mininet Hi-Fi, we were able to emulate such a topology and have our SDN load balancer schedule the client requests between the replicated servers and the multiple paths available to the servers.

The experiment consisted in having our load balancer perform using a combination of all server choice and path choice algorithms described in Section 3.3. Different experiments for different Δ values were run, and an optimal configuration was chosen. As each

client HTTP request exerted randomly different loads on the servers, we also varied the distribution that dictates these loads.

The main conclusion we take from all the experiments is the importance of the *Path Delay* algorithm. We were able to witness how scheduling traffic between available paths, taking network state into consideration, improves the load balancer performance. This result was consistently verified throughout all the experiments done. This attests to how significant the impact of SDNs can be in not only server load balancing, but in network management applications in general.

Our results have also shown that the best choice as server scheduling algorithm is *Round Robin*. In our basic scenario of an HTTP server cluster, *Round Robin*'s simplicity proved to be beneficial when comparing to the other "smarter" (but also heavier) algorithms.

Chapter 5

Conclusion

The root of the fragility and difficulty in managing today's networks lies in the complexity of the control and management planes. This is mainly due to decision logic and state being intertwined and embedded throughout all of the network's routing devices. Software-Defined Networking (SDN) is a new architecture in which network control is moved out of the individual routing devices to be implemented in software running on a logically centralized network controller.

We have developed a network control application that performs load balancing on a cluster of servers interconnected in a Software-Defined Network. Unlike current load balancing solutions, a software control application in an SDN environment has a high degree of customizability. During the course of this study we have experienced the ease to alter the configuration of our load balancer. Furthermore, current load balancing solutions, on top of being expensive pieces of hardware, are only able to schedule load among servers, and cannot choose the path taken by the traffic. This was shown to be an important advantage for the analyzed scenarios.

We have evaluated different load balancing algorithms using the Mininet Hi-Fi emulator. This network emulator's ability to easily prototype Software-Defined Networks in the constrained resources of a single computer machine makes it the *de facto* network emulator for SDN projects. By comparing the performance of the different load balancing algorithms, an important observation we were able to make was the benefit of scheduling load between multiple paths. In all configuration scenarios, the experiments have always shown a significant increase in performance when such scheduling was done

While our main goal was to build a control application that performs server load balancing on an SDN architecture, we were also able to attest to how simple it is to build and customize network control applications for such networks.

5.1 Future Work

On account of the delays suffered throughout our work, mainly due to the initial efforts in reasoning with Mininet and its lack of performance fidelity, we were not able to achieve a high degree of realism in our experiments. Two main things can be improved in order to augment the reliability of our findings.

1. Our experiments were tested using simple request service load distributions, in order to attempt to simulate some variability in our network and web service functionality. However, different distributions can be devised in order to make the experiment more realistic. Realism can be further improved by adding more distributions, such as client request arrival interval, or even running the experiment under real Web traces.

2. The topology used in our work is small and simple. It consists of five switches and three hosts. Even though larger topologies, with tens of switches and hosts have been shown to run under Mininet Hi-Fi, in the restrained resources of a single laptop it is infeasible to emulate datacenter topologies, which can have hundreds of switches. To further improve on our work, the experiments could be run in larger networks, for example, based on test beds such as GENI [13].

Even though in our simple scenario we were able to witness the benefits of using SDN in load balancing, these improvements to our experimental settings can further and more reliably attest to the practical advantages of this new network architecture.

The only path choice algorithm we devised that takes network load into account (*Path Delay*) simply tests the round trip time a packet takes in each path, and chooses the fastest one over an averaging window. However, a different algorithm could measure the available bandwidth on a path, and choose one that best fits to service a request.

All scheduling algorithms proposed in this work are reactive. They reactively assign client requests to the servers, by intercepting the first packet of a connection and installing individual flow entries that handle the remaining packets of a connection. However, it is possible to devise a proactive approach, in which forwarding rules are preloaded in the switches' flow tables when the system starts. An example of such an approach can be seen on [39], where different weights are assigned to the servers, and traffic is partitioned accordingly. Proactive scheduling has the advantage of avoiding the flow setup delay; however, they are not as dynamic and flexible as reactive rules can be.

List of Acronyms

ACL Access Control List

API Application Programming Interface

CDF Cummulative Distribution Function

CGI Common Gateway Interface

CPU Central Processor Unit

DPID Data Path Identifier

ECMP Equal-Cost Multi-Path

FC Flow Connections

HTTP HyperText Transfer Protocol

IP Internet Protocol

L Load

MAC Media Access Control

OS Operating System

OSI Open Systems Interconnection

OSPF Open Shortest Path First

PD Path Delay

QoS Quality of Service

RR Round Robin

RT Response Time

SC Server Connections

SDN Software-Defined Networking

SP Shortest Path

TCP Transmission Control Protocol

ToR Top of Rack

UDP User Datagram Protocol

VLAN Virtual Local Area Network

WAN Wide Area Network

Bibliography

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, October 2008.

[2] Mohammad Al-fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *The Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI'10)*, pages 19–19, San Jose, United States, April 2010.

[3] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *The proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC'10)*, pages 267–280, Melbourne, Australia, November 2010.

[4] Haakon Bryhni, Espen Klovning, and Øivind Kure. A Comparison of Load Balancing Techniques for Scalable Web Servers. *IEEE Network*, 14(4):58–64, Jul/Aug 2000.

[5] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *The 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI'05)*, pages 15–28, Boston, United States, May 2005.

[6] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, 3(3):28–39, May 1999.

[7] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *The 2007 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM'07)*, pages 1–12, Kyoto, Japan, August 2007.

[8] Martín Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *The 15th conference on Symposium on USENIX Security Symposium (USENIX-SS'05)*, volume 15, Vancouver, Canada, August 2006.

[9] Mark E. Corvella and Azer Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking (TON)*, 5(6):835–846, December 1997.

[10] Mark E. Corvella, Mor Harchol-Balter, and Cristina D. Murta. Task Assignment in a Distributed System: Improving Performance by Unbalancing Load. *ACM SIGMETRICS Performance Evaluation Review*, 26(1):268–269, June 1998.

[11] Wagner S. Dantas, Alysson N. Bessani, and Miguel Correia. Not Quickly, Just in Time: Improving the Timeliness and Reliability of Control Traffic in Utility Networks. In *Workshop on Hot Topics in System Dependability (HotDep'09)*, Lisbon, Portugal, June 2009.

[12] Project Floodlight. `http://www.projectfloodlight.org/floodlight/`.

[13] GENI - exploring network of the future. `http://www.geni.net`.

[14] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahini, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. *ACM SIGCOMM Computer Communication Review - SIGCOMM '09*, 39(4):51–62, October 2009.

[15] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhan. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM Computer Communication Review*, 35(5):41–54, October 2005.

[16] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, July 2008.

[17] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A Scalable and Fault-Tolerant Network Structure for Data Centers. *ACM SIGCOMM Computer Communication Review*, 38(8):75–86, October 2008.

[18] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-Based Emulation. In *The 8th international conference on Emerging Networking Experiments and Technologies (CoNEXT'12)*, pages 253–264, Nice, France, December 2012.

[19] Nikhil Handigol, Srini Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. Demo at ACM SIGCOMM'09, August 2009.

[20] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. Technical report, NextHop Technologies, November 2000. RFC 2992.

[21] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *The Proceedings of the ACM SIGCOMM 2013 conference (SIGCOMM '13)*, pages 3–14, Hong Kong, China, August 2013.

[22] Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. ONIX: A Distributed Control Platform for Large-scale Production Networks. In *The 9th USENIX conference on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, Canada, October 2010.

[23] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *The 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*, Monterey, United States, October 2010.

[24] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldman. Logically Centralized? State Distribution Trade-offs in Software Defined Networks. In *The first workshop on Hot topics in Software-Defined Networks*, pages 1–6, Helsinki, Finland, August 2012.

[25] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovations in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.

[26] Murphy McCauley, *NOX, POX and Controllers Galore*, Interview by Roy Chua. SDN Central - `http://www.sdncentral.com`.

[27] Radhika N. Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, and Vikram Subramanya. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. *ACM SIGCOMM Computer Communication Review - SIGCOMM '09*, 39(4):39–50, October 2009.

[28] The Network Simulator - ns2. `http://www.isi.edu/nsnam/ns/`.

[29] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.0*, June 2012.

[30] Open Networking Foundation. *Software-Defined Networking: The New Norm for Networks*, April 2012. ONF White Paper.

[31] Abhinav Pathak, Himabindu Pucha, Ying Zhang, Ying C. Hu, and Zhuoqing M. Mao. A Measurement Study of Internet Delay Asymmetry. In *The Proceedings of the 9th international conference on Passive and Active Network Measurement (PAM'08)*, pages 182–191, Cleveland, United States, April 2008.

[32] POX - An OpenFlow Controller. `http://www.noxrepo.org/pox`.

[33] Ryu SDN Controller. `http://osrg.github.io/ryu/`.

[34] Trevor Schroeder, Steve Goddard, and Byrav Ramamurthy. Scalable Web Server Clustering Technologies. *IEEE Network: The Magazine of Global Internetworking*, 14(3):38–45, May 2000.

[35] Yaron Schwartz, Yuval Shavitt, and Udi Weinsberg. A Measurement Study of the Origins of End-to-End Delay Variations. In *The Proceedings of the 9th international conference on Passive and Active Network Measurement (PAM'08)*, pages 182–191, Cleveland, United States, April 2008.

[36] Saeed Sharifian, Seyed A. Motamedi, and Mohammad K. Akbari. An Approximation-based Load-Balancing Algorithm with Admission Control for Cluster Web Servers with Dynamic Workloads. *The Journal of Supercomputing*, 53(3):440–463, September 2010.

[37] Scott Shenker. The future of networks, and the past of protocols. Open Networking Summit, October 2011. Keynote.

[38] Yong Meng Teo and Rassul Ayani. Comparison of Load Balancing Strategies on Cluster-based Web Servers. *Transactions of the Society for Modeling and Simulation*, 77:185–195, November 2001.

[39] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *The 11th USENIX conference on Hot topics in Management of Internet, Cloud and Enterprise Networks and Services (Hot-ICE'11)*, pages 12–12, Boston, United States, March 2011.

[40] Soheil H. Yeganeh, Amin Tootoochian, and Yashar Ganjali. On Scalability of Software-Defined Networking. *IEEE Communications Magazine*, 51(2):136–141, February 2013.